目錄

`Spark GraphX` 源码分析

`Spark GraphX` 是一个新的 `Spark API` ，它用于图和分布式图(`graph-parallel`)的计算。 `GraphX` 综合了 `Pregel` 和 `GraphLab` 两者的优点，即接口相对简单，又保证性能，可以应对点分割的图存储模式，胜任符合幂律分布的自然图的大型计算。本专题会详细介绍 `GraphX` 的实现原理，并对 `GraphX` 的存储结构以及部分操作作详细分析。

本专题介绍的内容如下：

目录

分布式图计算

在介绍 `GraphX` 之前，我们需要先了解分布式图计算框架。简言之，分布式图框架就是将大型图的各种操作封装成接口，让分布式存储、并行计算等复杂问题对上层透明，从而使工程师将焦点放在图相关的模型设计和使用上，而不用关心底层的实现细节。 分布式图框架的实现需要考虑两个问题，第一是怎样切分图以更好的计算和保存；第二是采用什么图计算模型。下面分别介绍这两个问题。

1 图切分方式

图的切分总体上说有点切分和边切分两种方式。

- 点切分：通过点切分之后，每条边只保存一次，并且出现在同一台机器上。邻居多的点会被分发到不同的节点上，增加了存储空间，并且有可能产生同步问题。但是，它的优点是减少了网络通信。

- 边切分：通过边切分之后，顶点只保存一次，切断的边会打断保存在两台机器上。在基于边的操作时，对于两个顶点分到两个不同的机器的边来说，需要进行网络传输数据。这增加了网络传输的数据量，但好处是节约了存储空间。

以上两种切分方式虽然各有优缺点，但是点切分还是占有优势。 `GraphX` 以及后文提到的 `Pregel` 、 `GraphLab` 都使用到了点切分。

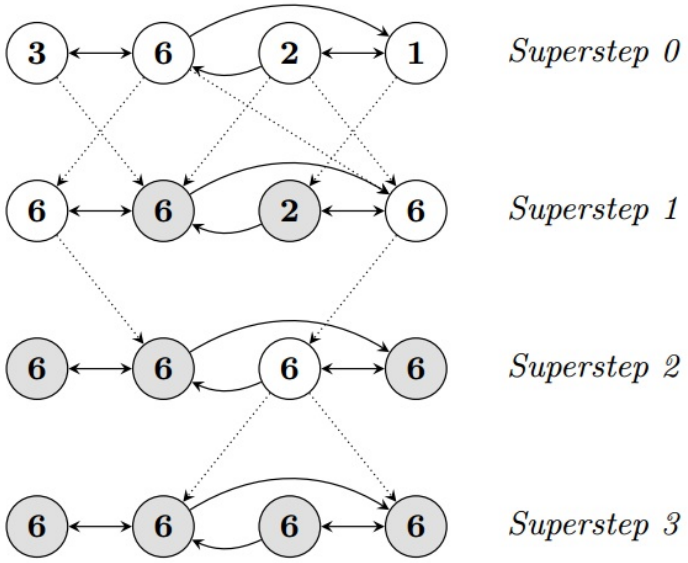
2 图计算框架

图计算框架基本上都遵循分布式批同步（ `Bulk Synchronous Parallell,BSP` ）计算模式。基于 `BSP` 模式，目前有两种比较成熟的图计算框架: `Pregel` 框架和 `GraphLab` 框架。

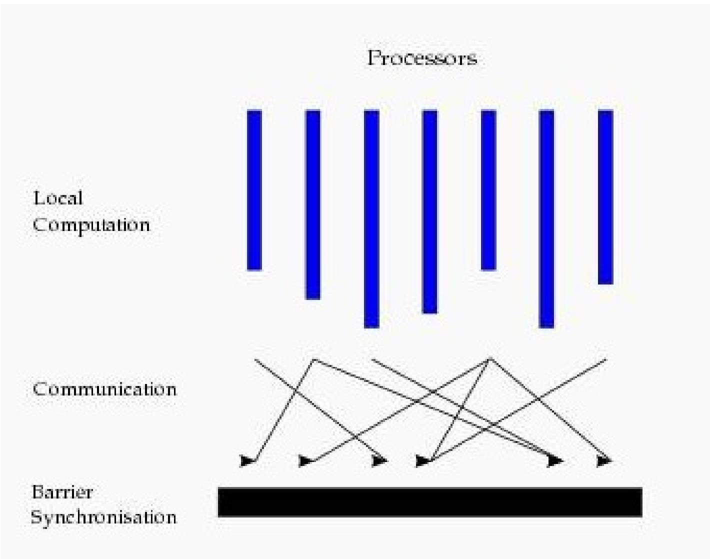
2.1 BSP

2.1.1 BSP基本原理

在 `BSP` 中，一次计算过程由一系列全局超步组成，每一个超步由并发计算、通信和同步三个步骤组成。同步完成，标志着这个超步的完成及下一个超步的开始。 `BSP` 模式的准则是批量同步(`bulk synchrony`)，其独特之处在于超步(`superstep`)概念的引入。一个 `BSP` 程序同时具有水平和垂直两个方面的结构。从垂直上看,一个 `BSP` 程序由一系列串行的超步(`superstep`)组成,如图所示:



从水平上看，在一个超步中，所有的进程并行执行局部计算。一个超步可分为三个阶段，如图所示:



- 本地计算阶段，每个处理器只对存储在本地内存中的数据进行本地计算。
- 全局通信阶段，对任何非本地数据进行操作。
- 栅栏同步阶段，等待所有通信行为的结束。

2.1.2 BSP模型特点

BSP模型有如下几个特点：

- 1 将计算划分为一个一个的超步(`superstep`)，有效避免死锁;
- 2 将处理器和路由器分开，强调了计算任务和通信任务的分开，而路由器仅仅完成点到点的消息传递，不提供组合、复制和广播等功能，这样做既掩盖具体的互连网络拓扑，又简化了通信协议；
- 3 采用障碍同步的方式、以硬件实现的全局同步是可控的粗粒度级，提供了执行紧耦合同步式并行算法的有效方式

2.2 `Pregel` 框架

`Pregel` 是一种面向图算法的分布式编程框架，采用迭代的计算模型：在每一轮，每个顶点处理上一轮收到的消息，并发出消息给其它顶点，并更新自身状态和拓扑结构（出、入边）等。

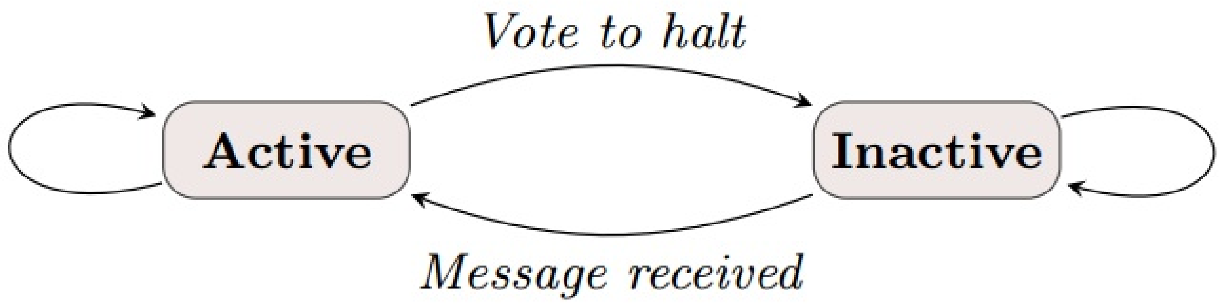
2.2.1 `Pregel` 框架执行过程

在 `Pregel` 计算模式中，输入是一个有向图，该有向图的每一个顶点都有一个相应的由字符串描述的 `vertex identifier` 。每一个顶点都有一些属性，这些属性可以被修改，其初始值由用户定义。每一条有向边都和其源顶点关联，并且也拥有一些用户定义的属性和值，并同时还记录了其目的顶点的 `ID` 。

一个典型的 `Pregel` 计算过程如下：读取输入，初始化该图，当图被初始化好后，运行一系列的超步，每一次超步都在全局的角度上独立运行，直到整个计算结束，输出结果。 在每一次超步中，顶点的计算都是并行的，并且执行用户定义的同一个函数。每个顶点可以修改其自身的状态信息或以它为起点的出边的信息，从前序超步中接受消息，并传送给其后续超步，或者修改整个图的拓扑结构。边，在这种计算模式中并不是核心对象，没有相应的计算运行在其上。

算法是否能够结束取决于是否所有的顶点都已经 `vote` 标识其自身已经达到 `halt` 状态了。在 `superstep 0` 中，所有顶点都置于 `active` 状态，每一个 `active` 的顶点都会在计算的执行中在某一次的 `superstep` 中被计算。顶点通过将其自身的状态设置成 `halt` 来表示它已经不再 `active` 。这就表示该顶点没有进一步的计算需要进行，除非被其他的运算触发，而 `Pregel` 框架将不会在接下

来的 `superstep` 中计算该顶点，除非该顶点收到一个其他 `superstep` 传送的消息。 如果顶点接收到消息，该消息将该顶点重新置 `active` ，那么在随后的计算中该顶点必须再次 `deactive` 其自身。整个计算在所有顶点都达到 `inactive` 状态，并且没有消息在传送的时候宣告结束。这种简单的状态机制在下图中描述：



我们用 `PageRank` 为例来说明 `Pregel` 的计算过程。

```
def PageRank(v: Id, msgs: List[Double]) {
// 计算消息和
var msgSum = 0
for (m <- msgs) { msgSum = msgSum + m }
// 更新 PageRank (PR)
A(v).PR = 0.15 + 0.85 * msgSum
// 广播新的PR消息
for (j <- OutNbrs(v)) {
msg = A(v).PR / A(v).NumLinks
send_msg(to=j, msg)
}
// 检查终止
if (converged(A(v).PR)) voteToHalt(v)
}
```

以上代码中，顶点 `v` 首先接收来自上一次迭代的消息，计算它们的和。然后使用计算的消息和重新计算 `PageRank` ，之后程序广播这个重新计算的 `PageRank` 的值到顶点 `v` 的所有邻居，最后程序判断算法是否应该停止。

2.2.1 `Pregel` 框架的消息模式

`Pregel` 选择了一种纯消息传递的模式，忽略远程数据读取和其他共享内存的方式，这样做有两个原因。

- 第一，消息的传递有足够高效的表达能力，不需要远程读取（ `remote reads` ）。
- 第二，性能的考虑。在一个集群环境中，从远程机器上读取一个值是会有很高的延迟的，这种情况很难避免。而消息传递模式通过异步和批量的方式传递消息，可以缓解这种远程读取的延迟。

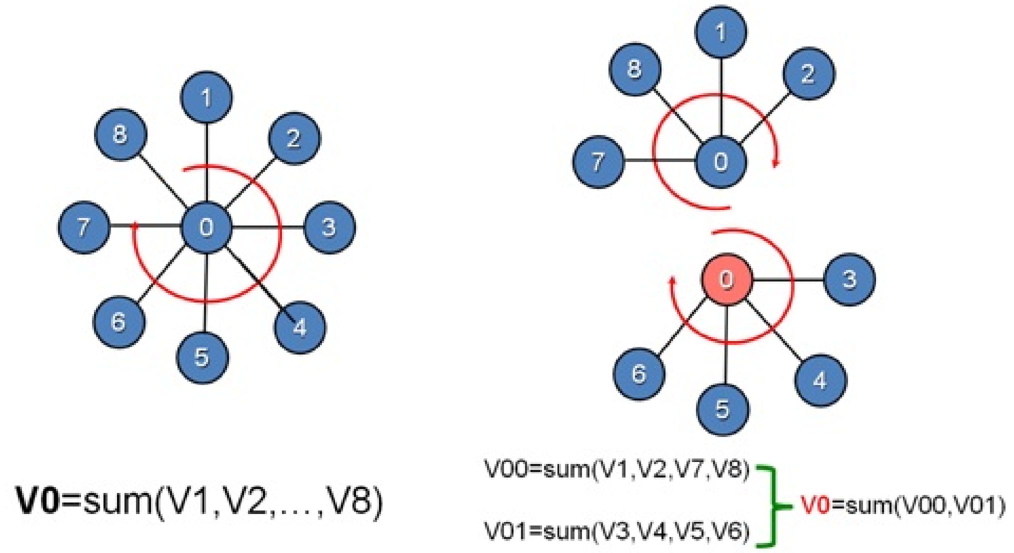
图算法其实也可以被写成是一系列的链式 `MapReduce` 作业。选择不同的模式的原因在于可用性和性能。 `Pregel` 将顶点和边在本地机器进行运算，而仅仅利用网络来传输信息，而不是传输数据。 而 `MapReduce` 本质上是面向函数的，所以将图算法用 `MapReduce` 来实现就需要将整个图的状态从一个阶段传输到另外一个阶段，这样就需要许多的通信和随之而来的序列化和反序列化的开销。另外，在一连串的 `MapReduce` 作业中各阶段需要协同工作也给编程增加了难度，这样的情况能够在 `Pregel` 的各轮超步的迭代中避免。

2.2.3 `Pregel` 框架的缺点

这个模型虽然简单，但是缺陷明显，那就是对于邻居数很多的顶点，它需要处理的消息非常庞大，而且在这个模式下，它们是无法被并发处理的。所以对于符合幂律分布的自然图，这种计算模型下很容易发生假死或者崩溃。

2.3 `GraphLab` 框架

`GraphLab` 将数据抽象成 `Graph` 结构，将基于顶点切分的算法的执行过程抽象成 `Gather`、`Apply`、`Scatter` 三个步骤。以下面的例子作为一个说明。



示例中，需要完成对 `V0` 邻接顶点的求和计算，串行实现中， `V0` 对其所有的邻接点进行遍历，累加求和。而 `GraphLab` 中，将顶点 `V0` 进行切分，将 `V0` 的边关系以及对应的邻接点部署在两台处理器上，各台机器上并行进行部分求和运算，然后通过 `master` （蓝色）顶点和 `mirror` （橘红色）顶点的通信完成最终的计算。

2.3.1 `GraphLab` 框架的数据模型

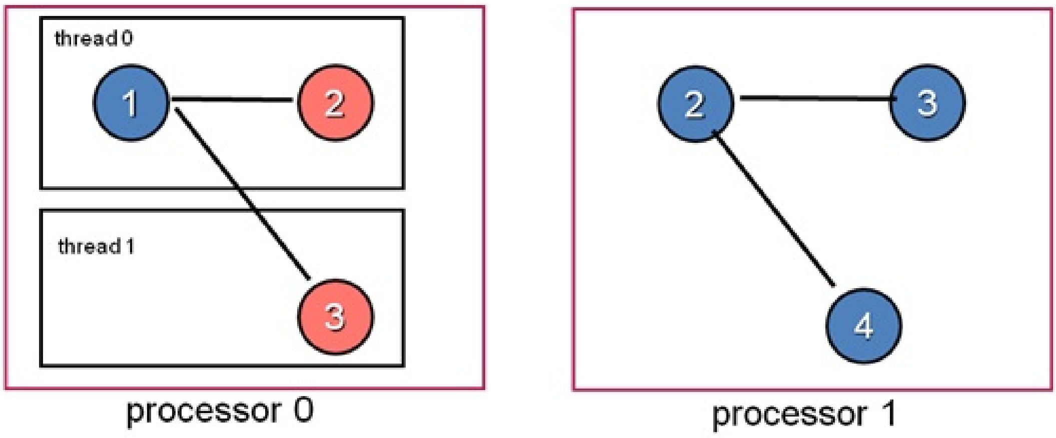
对于分割的某个顶点，它会被部署到多台机器，一台机器作为 `master` 顶点，其余机器作为 `mirror` 。 `master` 作为所有 `mirror` 的管理者，负责给 `mirror` 安排具体计算任务; `mirror` 作为该顶点在各台机器上的代理执行者，与 `master` 数据的保持同步。

对于某条边， `GraphLab` 将其唯一部署在某一台机器上，而对边关联的顶点进行多份存储，解决了边数据量大的问题。

同一台机器上的所有顶点和边构成一个本地图（ `local graph）` ,在每台机器上，存在一份本地 `id` 到全局 `id` 的映射表。顶点是一个进程上所有线程共享的，在并行计算过程中，各个线程分摊进程中所有顶点的 `gather->apply->scatter` 操作。

我们用下面这个例子说明， `GraphLab` 是怎么构建 `Graph` 的。图中，以顶点 `v2` 和 `v3` 进行分割。顶点 `v2` 和 `v3` 同时存在于两个进程中，并且两个线程共同分担顶点计算。





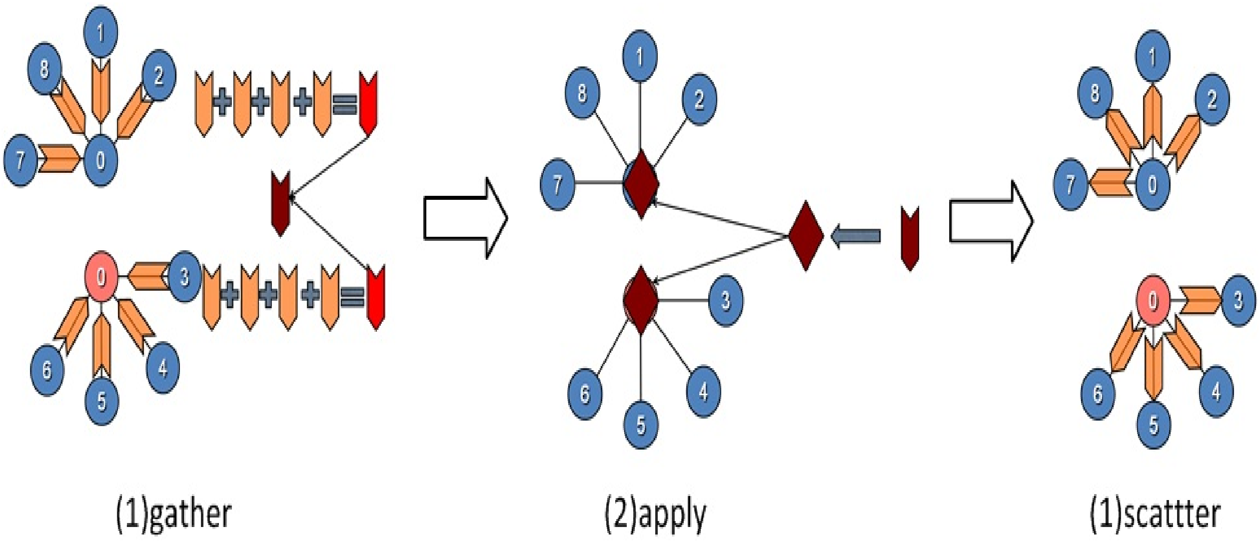
2.3.2 `GraphLab` 框架的执行模型

每个顶点每一轮迭代会经过 `gather -> apple -> scatter` 三个阶段。

- **Gather**阶段，工作顶点的边从连接顶点和自身收集数据。这一阶段对工作顶点、边都是只读的。
- **Apply**阶段， `mirror` 将 `gather` 阶段计算的结果发送给 `master` 顶点， `master` 进行汇总并结合上一步的顶点数据，按照业务需求进行进一步的计算，然后更新 `master` 的顶点数据，并同步给 `mirror` 。 `Apply` 阶段中，工作顶点可修改，边不可修改。
- **Scatter**阶段，工作顶点更新完成之后，更新边上的数据，并通知对其有依赖的邻结顶点更新状态。在 `scatter` 过程中，工作顶点只读，边上数据可写。

在执行模型中， `GraphLab` 通过控制三个阶段的读写权限来达到互斥的目的。在 `gather` 阶段只读， `apply` 对顶点只写， `scatter` 对边只写。并行计算的同步通过 `master` 和 `mirror` 来实现， `mirror` 相当于每个顶点对外的一个接口人，将复杂的数据通信抽象成顶点的行为。

下面这个例子说明 `GraphLab` 的执行模型：



利用 GraphLab 实现的 PageRank 的代码如下所示：

```
//汇总
def Gather(a: Double, b: Double) = a + b
//更新顶点
def Apply(v, msgSum) {
A(v).PR = 0.15 + 0.85 * msgSum
if (converged(A(v).PR)) voteToHalt(v)
}
//更新边
def Scatter(v, j) = A(v).PR / A(v).NumLinks
```

由于 gather/scatter 函数是以单条边为操作粒度，所以对于一个顶点的众多邻边，可以分别由相应的节点独立调用 gather/scatter 函数。这一设计主要是为了适应点分割的图存储模式，从而避免 Pregel 模型会遇到的问题。

3 GraphX

GraphX 也是基于 BSP 模式。GraphX 公开了一个类似 Pregel 的操作，它是广泛使用的 Pregel 和 GraphLab 抽象的一个融合。

在 GraphX 中，Pregel 操作者执行一系列的超步，在这些超步中，顶点从之前的超步中接收进入(inbound)消息，为顶点属性计算一个新的值，然后在以后的超步中发送消息到邻居顶点。不像 Pregel 而更像 GraphLab ，消息通过

边 `triplet` 的一个函数被并行计算，消息的计算既会访问源顶点特征也会访问目的顶点特征。在超步中，没有收到消息的顶点会被跳过。当没有消息遗留时， `Pregel` 操作停止迭代并返回最终的图。

4 参考文献

【1】Preg el: A System for Larg e-Scale Graph Processing

【2】快刀初试：Spark GraphX在淘宝的实践

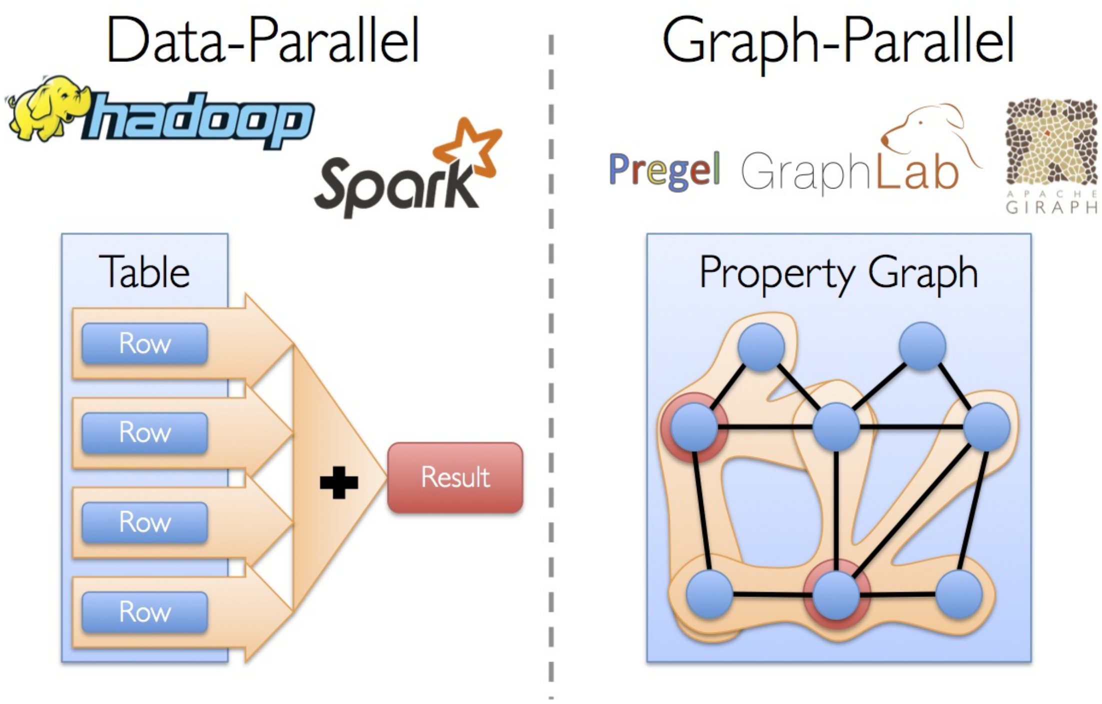
【3】GraphLab:A New Parallel Framework for Machine Learning

GraphX介绍

1 GraphX的优势

`GraphX` 是一个新的 `Spark API` ，它用于图和分布式图(`graph-parallel`)的计算。 `GraphX` 通过引入弹性分布式属性图（Resilient Distributed Property Graph）： 顶点和边均有属性的有向多重图，来扩展 `Spark RDD` 。为了支持图计算， `GraphX` 开发了一组基本的功能操作以及一个优化过的 `Pregel API` 。另外， `GraphX` 包含了一个快速增长的图算法和图 `builders` 的 集合，用以简化图分析任务。

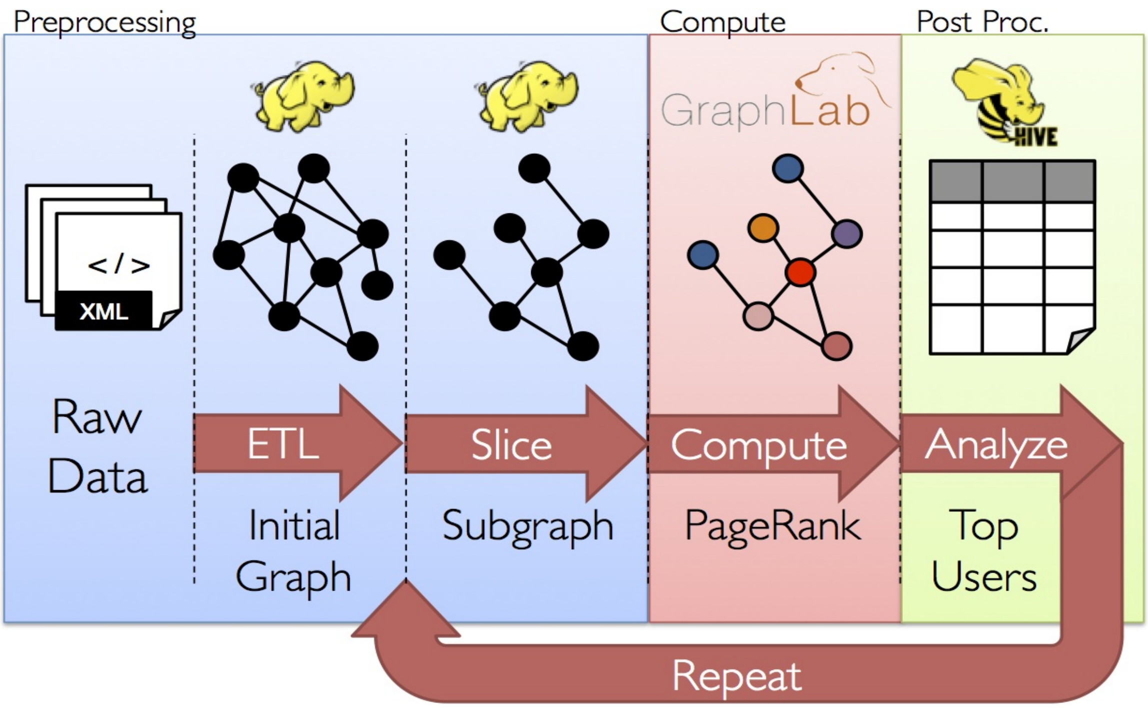
从社交网络到语言建模，不断增长的规模以及图形数据的重要性已经推动了许多新的分布式图系统（如Giraph和GraphLab）的发展。 通过限制计算类型以及引入新的技术来切分和分配图，这些系统可以高效地执行复杂的图形算法，比一般的分布式数据计算（ `data-parallel` ，如 `spark` 、 `MapReduce` ）快很多。



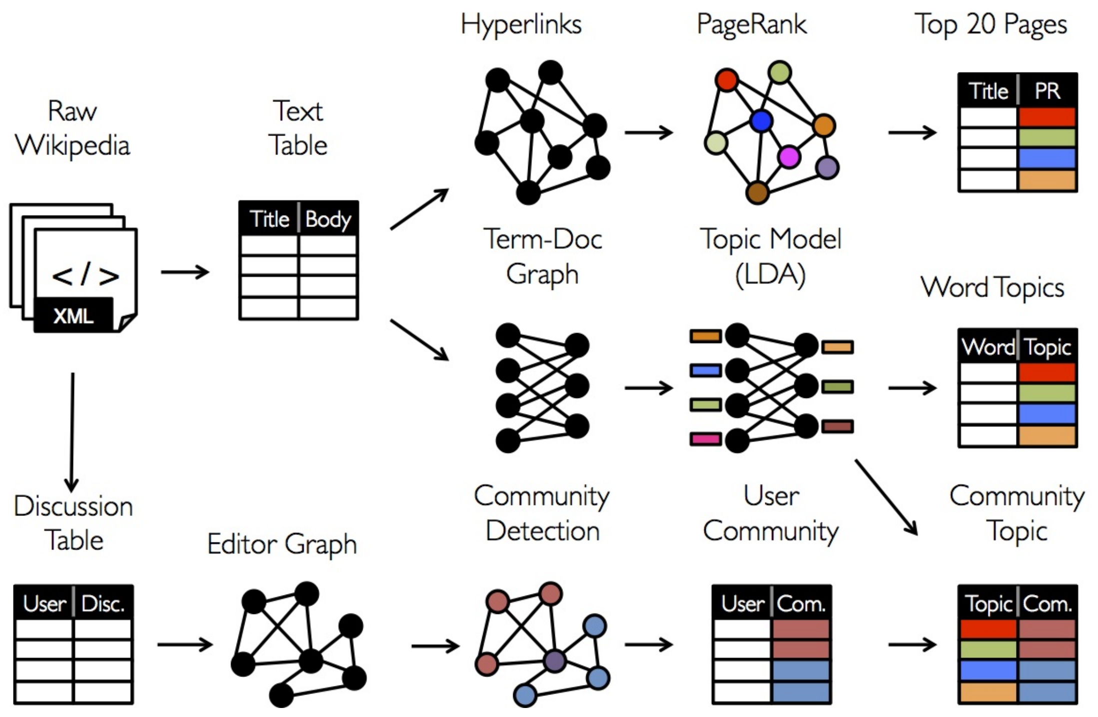
分布式图（ `graph-parallel` ）计算和分布式数据（ `data-parallel` ）计算类似，分布式数据计算采用了一种 `record-centric` 的集合视图，而分布式图计算采用了一种 `vertex-centric` 的图视图。 分布式数据计算通过同时处理独立

的数据来获得并发的目的，分布式图计算则是通过对图数据进行分区（即切分）来获得并发的目的。更准确的说，分布式图计算递归地定义特征的转换函数（这种转换函数作用于邻居特征），通过并发地执行这些转换函数来获得并发的目的。

分布式图计算比分布式数据计算更适合图的处理，但是在典型的图处理流水线中，它并不能很好地处理所有操作。例如，虽然分布式图系统可以很好的计算 `PageRank` 以及 `label diffusion` ，但是它们不适合从不同的数据源构建图或者跨过多个图计算特征。 更准确的说，分布式图系统提供的更窄的计算视图无法处理那些构建和转换图结构以及跨越多个图的需求。分布式图系统中无法提供的这些操作需要数据在图本体之上移动并且需要一个图层面而不是单独的顶点或边层面的计算视图。例如，我们可能想限制我们的分析到几个子图上，然后比较结果。 这不仅需要改变图结构，还需要跨多个图计算。



我们如何处理数据取决于我们的目标，有时同一原始数据可能会处理成许多不同表和图的视图，并且图和表之间经常需要能够相互移动。如下图所示：



所以我们的图流水线必须通过组合 `graph-parallel` 和 `data- parallel` 来实现。但是这种组合必然会导致大量的数据移动以及数据复制，同时这样的系统也非常复杂。 例如，在传统的图计算流水线中，在 `Table View` 视图下，可能需要 `Spark` 或者 `Hadoop` 的支持，在 `Graph View` 这种视图下，可能需要 `Prege` 或者 `GraphLab` 的支持。也就是把图和表分在不同的系统中分别处理。不同系统之间数据的移动和通信会成为很大的负担。

`GraphX` 项目将 `graph-parallel` 和 `data-parallel` 统一到一个系统中，并提供了一个唯一的组合 `API` 。 `GraphX` 允许用户把数据当做一个图和一个集合（ `RDD` ），而不需要数据移动或者复制。也就是说 `GraphX` 统一了 `Graph View` 和 `Table View` ， 可以非常轻松的做 `pipeline` 操作。

2 弹性分布式属性图

`GraphX` 的核心抽象是弹性分布式属性图，它是一个有向多重图，带有连接到每个顶点和边的用户定义的对象。 有向多重图中多个并行的边共享相同的源和目的顶点。支持并行边的能力简化了建模场景，相同的顶点可能存在多种关系(例如 `co-worker` 和 `friend`)。 每个顶点用一个唯一的64位长的标识符（ `VertexID` ）作为 `key` 。 `GraphX` 并没有对顶点标识强加任何排序。同样，边拥有相应的源和目的顶点标识符。

属性图扩展了 `Spark RDD` 的抽象，有 `Table` 和 `Graph` 两种视图，但是只需要一份物理存储。两种视图都有自己独有的操作符，从而使我们同时获得了操作的灵活性和执行的高效率。 属性图以 `vertex(VD)` 和 `edge(ED)` 类型作为参数类型，这些类型分别是顶点和边相关联的对象的类型。

在某些情况下，在同样的图中，我们可能希望拥有不同属性类型的顶点。这可以通过继承完成。例如，将用户和产品建模成一个二分图，我们可以用如下方式：

```
class VertexProperty()
case class UserProperty(val name: String) extends
VertexProperty
case class ProductProperty(val name: String, val price:
Double) extends VertexProperty
// The graph might then have the type:
var graph: Graph[VertexProperty, String] = null
```

和 `RDD` 一样，属性图是不可变的、分布式的、容错的。图的值或者结构的改变需要生成一个新的图来实现。注意，原始图中不受影响的部分都可以在新图中重用，用来减少存储的成本。 执行者使用一系列顶点分区方法来对图进行分区。如 `RDD` 一样，图的每个分区可以在发生故障的情况下被重新创建在不同的机器上。

逻辑上,属性图对应于一对类型化的集合(`RDD`),这个集合包含每一个顶点和边的属性。因此，图的类中包含访问图中顶点和边的成员变量。

```
class Graph[VD, ED] {
  val vertices: VertexRDD[VD]
  val edges: EdgeRDD[ED]
}
```

`VertexRDD[VD]` 和 `EdgeRDD[ED]` 类是 `RDD[(VertexID, VD)]` 和 `RDD[Edge[ED]]` 的继承和优化版本。 `VertexRDD[VD]` 和 `EdgeRDD[ED]` 都提供了额外的图计算功能并提供内部优化功能。

```
abstract class VertexRDD[VD](
    sc: SparkContext,
    deps: Seq[Dependency[_]]) extends RDD[(VertexId, VD)]
(sc, deps)

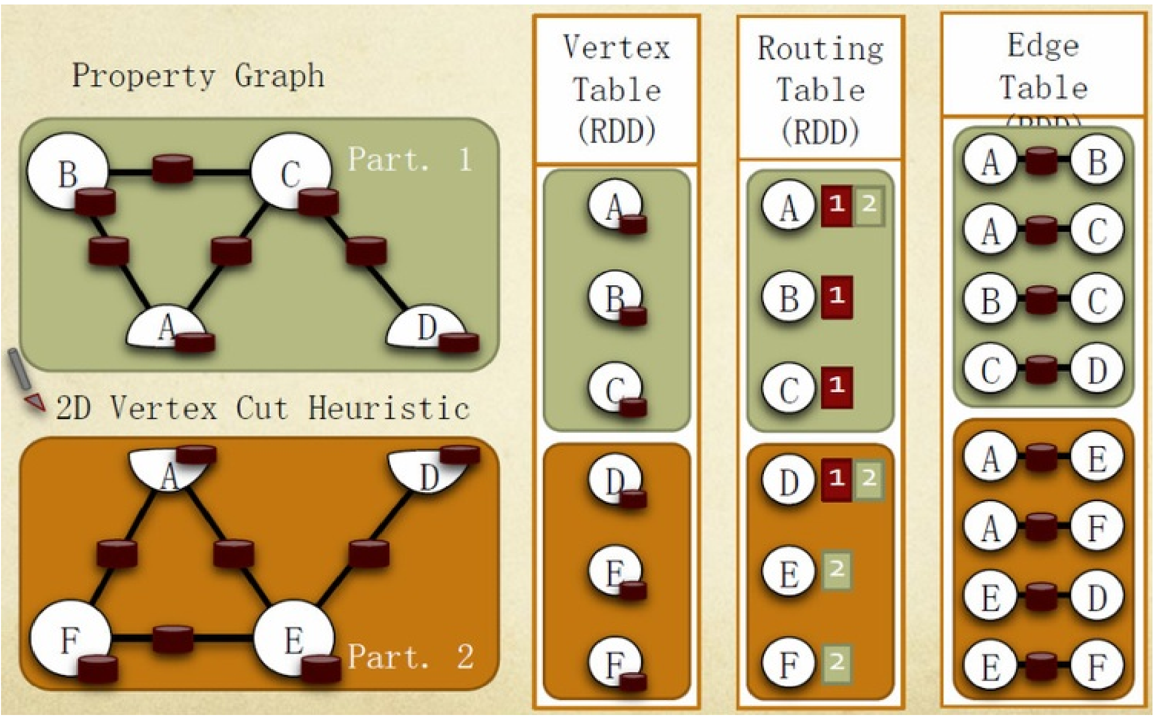
abstract class EdgeRDD[ED](
    sc: SparkContext,
    deps: Seq[Dependency[_]]) extends RDD[Edge[ED]](sc,
deps)
```

3 GraphX的图存储模式

`Graphx` 借鉴 `PowerGraph`，使用的是 `Vertex-Cut`（点分割）方式存储图，用三个 `RDD` 存储图数据信息：

- `VertexTable(id, data)`：`id` 为顶点 `id`，`data` 为顶点属性
- `EdgeTable(pid, src, dst, data)`：`pid` 为分区 `id`，`src` 为源顶点 `id`，`dst` 为目的顶点 `id`，`data` 为边属性
- `RoutingTable(id, pid)`：`id` 为顶点 `id`，`pid` 为分区 `id`

点分割存储实现如下图所示：



在后文的图构建部分，我们会详细介绍这三个部分。

4 GraphX底层设计的核心点

- 1 对 `Graph` 视图的所有操作，最终都会转换成其关联的 `Table` 视图的 `RDD` 操作来完成。一个图的计算在逻辑上等价于一系列 `RDD` 的转换过程。因此， `Graph` 最终具备了 `RDD` 的3个关键特性：不变性、分布性和容错性。其中最关键的是不变性。逻辑上，所有图的转换和操作都产生了一个新图；物理上， `GraphX` 会有一定程度的不变顶点和边的复用优化，对用户透明。
- 2 两种视图底层共用的物理数据，由 `RDD[VertexPartition]` 和 `RDD[EdgePartition]` 这两个 `RDD` 组成。点和边实际都不是以表 `Collection[tuple]` 的形式存储的，而是由 `VertexPartition/EdgePartition` 在内部存储一个带索引结构的分片数据块，以加速不同视图下的遍历速度。不变的索引结构在 `RDD` 转换过程中是共用的，降低了计算和存储开销。
- 3 图的分布式存储采用点分割模式，而且使用 `partitionBy` 方法，由用户指定不同的划分策略。下一章会具体讲到划分策略。

5 参考文献

【1】spark graphx参考文献

【2】快刀初试：Spark GraphX在淘宝的实践

【3】GraphX: Unifying Data-Parallel and Graph-Parallel

点分割存储

在第一章分布式图系统中，我们介绍了图存储的两种方式：点分割存储和边分割存储。`GraphX` 借鉴 `powerGraph` ，使用的是点分割方式存储图。这种存储方式特点是任何一条边只会出现在一台机器上，每个点有可能分布到不同的机器上。当点被分割到不同机器上时，是相同的镜像，但是有一个点作为主点,其他的点作为虚点，当点的数据发生变化时,先更新主点的数据，然后将所有更新好的数据发送到虚点所在的所有机器，更新虚点。 这样做的好处是在边的存储上是没有冗余的，而且对于某个点与它的邻居的交互操作，只要满足交换律和结合律，就可以在不同的机器上面执行，网络开销较小。但是这种分割方式会存储多份点数据，更新点时，会发生网络传输，并且有可能出现同步问题。

`GraphX` 在进行图分割时，有几种不同的分区(`partition`)策略，它通过 `PartitionStrategy` 专门定义这些策略。在 `PartitionStrategy` 中，总共定义了 `EdgePartition2D` 、 `EdgePartition1D` 、 `RandomVertexCut` 以及 `CanonicalRandomVertexCut` 这四种不同的分区策略。下面分别介绍这几种策略。

1 RandomVertexCut

```
case object RandomVertexCut extends PartitionStrategy {
    override def getPartition(src: VertexId, dst:
VertexId, numParts: PartitionID): PartitionID = {
      math.abs((src, dst).hashCode()) % numParts
    }
  }
```

这个方法比较简单，通过取源顶点和目标顶点 `id` 的哈希值来将边分配到不同的分区。这个方法会产生一个随机的边分割，两个顶点之间相同方向的边会分配到同一个分区。

2 CanonicalRandomVertexCut

```
case object CanonicalRandomVertexCut extends
PartitionStrategy {
    override def getPartition(src: VertexId, dst:
VertexId, numParts: PartitionID): PartitionID = {
      if (src < dst) {
        math.abs((src, dst).hashCode()) % numParts
      } else {
        math.abs((dst, src).hashCode()) % numParts
      }
    }
  }
```

这种分割方法和前一种方法没有本质的不同。不同的是，哈希值的产生带有确定的方向（即两个顶点中较小 `id` 的顶点在前）。两个顶点之间所有的边都会分配到同一个分区，而不管方向如何。

3 EdgePartition1D

```
case object EdgePartition1D extends PartitionStrategy {
    override def getPartition(src: VertexId, dst:
VertexId, numParts: PartitionID): PartitionID = {
      val mixingPrime: VertexId = 1125899906842597L
      (math.abs(src * mixingPrime) % numParts).toInt
    }
  }
```

这种方法仅仅根据源顶点 `id` 来将边分配到不同的分区。有相同源顶点的边会分配到同一分区。

4 EdgePartition2D

```
case object EdgePartition2D extends PartitionStrategy {
    override def getPartition(src: VertexId, dst:
VertexId, numParts: PartitionID): PartitionID = {
      val ceilSqrtNumParts: PartitionID =
math.ceil(math.sqrt(numParts)).toInt
      val mixingPrime: VertexId = 1125899906842597L
      if (numParts == ceilSqrtNumParts *
ceilSqrtNumParts) {
        // Use old method for perfect squared to ensure
we get same results
        val col: PartitionID = (math.abs(src *
mixingPrime) % ceilSqrtNumParts).toInt
        val row: PartitionID = (math.abs(dst *
mixingPrime) % ceilSqrtNumParts).toInt
        (col * ceilSqrtNumParts + row) % numParts
      } else {
        // Otherwise use new method
        val cols = ceilSqrtNumParts
        val rows = (numParts + cols - 1) / cols
        val lastColRows = numParts - rows * (cols - 1)
        val col = (math.abs(src * mixingPrime) % numParts
/ rows).toInt
        val row = (math.abs(dst * mixingPrime) % (if (col
< cols - 1) rows else lastColRows)).toInt
        col * rows + row
      }
    }
  }
```

这种分割方法同时使用到了源顶点 `id` 和目的顶点 `id` 。它使用稀疏边连接矩阵的2维区分来将边分配到不同的分区，从而保证顶点的备份数不大于 `2 * sqrt(numParts)` 的限制。这里 `numParts` 表示分区数。 这个方法的实现分两种情况，即分区数能完全开方和不能完全开方两种情况。当分区数能完全开方时，采用下面的方法：

```
 val col: PartitionID = (math.abs(src * mixingPrime) %
ceilSqrtNumParts).toInt
 val row: PartitionID = (math.abs(dst * mixingPrime) %
ceilSqrtNumParts).toInt
 (col * ceilSqrtNumParts + row) % numParts
```

当分区数不能完全开方时，采用下面的方法。这个方法的最后一列允许拥有不同的行数。

```
val cols = ceilSqrtNumParts
val rows = (numParts + cols - 1) / cols
//最后一列允许不同的行数
val lastColRows = numParts - rows * (cols - 1)
val col = (math.abs(src * mixingPrime) % numParts /
rows).toInt
val row = (math.abs(dst * mixingPrime) % (if (col < cols
- 1) rows else lastColRows)).toInt
col * rows + row
```

下面举个例子来说明该方法。假设我们有一个拥有12个顶点的图，要把它切分到9台机器。我们可以用下面的稀疏矩阵来表示:

```
          __________________________________
v0     | P0 *       | P1         | P2    *   |
v1     |  ****      |  *         |           |
v2     |  *******   |      **    |  ****     |
v3     |  *****     |  *  *      |       *   |
          ----------------------------------
v4     | P3 *       | P4 ***     | P5 **   * |
v5     |  *  *      |  *         |           |
v6     |       *    |      **    |  ****     |
v7     |  * * *     |  *  *      |       *   |
          ----------------------------------
v8     | P6   *     | P7     *   | P8  *    *|
v9     |     *      |  *     *   |           |
v10    |       *    |      **    |  *  *     |
v11    | * <-E      |  ***       |       **  |
          ----------------------------------
```

上面的例子中 `*` 表示分配到处理器上的边。 `E` 表示连接顶点 `v11` 和 `v1` 的边，它被分配到了处理器 `P6` 上。为了获得边所在的处理器，我们将矩阵切分为 `sqrt(numParts) * sqrt(numParts)` 块。 注意，上图中与顶点 `v11` 相连接的边只出现在第一列的块 `(P0,P3,P6)` 或者最后一行的块 `(P6,P7,P8)` 中，这保证了 `V11` 的副本数不会超过 `2 * sqrt(numParts)` 份，在上例中即副本不能超过6份。

在上面的例子中， `P0` 里面存在很多边，这会造成工作的不均衡。为了提高均衡，我们首先用顶点 `id` 乘以一个大的素数，然后再 `shuffle` 顶点的位置。乘以一个大的素数本质上不能解决不平衡的问题，只是减少了不平衡的情况发生。

5 参考文献

【1】spark源码

GraphX 中 vertices 、 edges 以及 triplets

vertices 、 edges 以及 triplets 是 GraphX 中三个非常重要的概念。我们在前文GraphX介绍中对这三个概念有初步的了解。

1 vertices

在 GraphX 中， vertices 对应着名称为 VertexRDD 的 RDD 。这个 RDD 有顶点 id 和顶点属性两个成员变量。它的源码如下所示：

```
abstract class VertexRDD[VD](
    sc: SparkContext,
    deps: Seq[Dependency[_]]) extends RDD[(VertexId, VD)]
(sc, deps)
```

从源码中我们可以看到， VertexRDD 继承自 RDD[(VertexId, VD)] ，这里 VertexId 表示顶点 id ， VD 表示顶点所带的属性的类别。这从另一个角度也说明 VertexRDD 拥有顶点 id 和顶点属性。

2 edges

在 GraphX 中， edges 对应着 EdgeRDD 。这个 RDD 拥有三个成员变量，分别是源顶点 id 、目标顶点 id 以及边属性。它的源码如下所示：

```
abstract class EdgeRDD[ED](
    sc: SparkContext,
    deps: Seq[Dependency[_]]) extends RDD[Edge[ED]](sc,
deps)
```

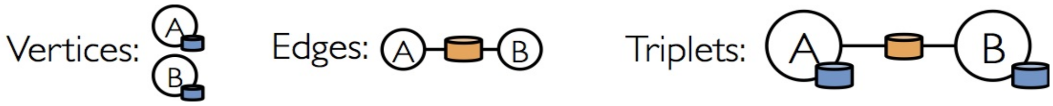
从源码中我们可以看到， EdgeRDD 继承自 RDD[Edge[ED]] ，即类型为 Edge[ED] 的 RDD 。 Edge[ED] 在后文会讲到。

3 triplets

在 `GraphX` 中， `triplets` 对应着 `EdgeTriplet` 。它是一个三元组视图，这个视图逻辑上将顶点和边的属性保存为一个 `RDD[EdgeTriplet[VD, ED]]` 。可以通过下面的 `Sql` 表达式表示这个三元视图的含义:

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS
dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

同样，也可以通过下面图解的形式来表示它的含义：



`EdgeTriplet` 的源代码如下所示：

```
class EdgeTriplet[VD, ED] extends Edge[ED] {
  //源顶点属性
  var srcAttr: VD = _ // nullValue[VD]
  //目标顶点属性
  var dstAttr: VD = _ // nullValue[VD]
  protected[spark] def set(other: Edge[ED]):
EdgeTriplet[VD, ED] = {
    srcId = other.srcId
    dstId = other.dstId
    attr = other.attr
    this
  }
```

`EdgeTriplet` 类继承自 `Edge` 类，我们来看看这个父类：

```
case class Edge[@specialized(Char, Int, Boolean, Byte,
Long, Float, Double) ED] (
    var srcId: VertexId = 0,
    var dstId: VertexId = 0,
    var attr: ED = null.asInstanceOf[ED])
  extends Serializable
```

`Edge` 类中包含源顶点 `id` ，目标顶点 `id` 以及边的属性。所以从源代码中我们可以知道， `triplets` 既包含了边属性也包含了源顶点的 `id` 和属性、目标顶点的 `id` 和属性。

4 参考文献

【1】spark源码

图的构建

`GraphX` 的 `Graph` 对象是用户操作图的入口。前面的章节我们介绍过，它包含了边(`edges`)、顶点(`vertices`)以及 `triplets` 三部分，并且这三部分都包含相应的属性，可以携带额外的信息。

1 构建图的方法

构建图的入口方法有两种，分别是根据边构建和根据边的两个顶点构建。

- 根据边构建图**(Graph.fromEdges)**

```
def fromEdges[VD: ClassTag, ED: ClassTag](
      edges: RDD[Edge[ED]],
      defaultValue: VD,
      edgeStorageLevel: StorageLevel =
StorageLevel.MEMORY_ONLY,
      vertexStorageLevel: StorageLevel =
StorageLevel.MEMORY_ONLY): Graph[VD, ED] = {
    GraphImpl(edges, defaultValue, edgeStorageLevel,
vertexStorageLevel)
  }
```

- 根据边的两个顶点数据构建**(Graph.fromEdgeTuples)**

```
 def fromEdgeTuples[VD: ClassTag](
      rawEdges: RDD[(VertexId, VertexId)],
      defaultValue: VD,
      uniqueEdges: Option[PartitionStrategy] = None,
      edgeStorageLevel: StorageLevel =
StorageLevel.MEMORY_ONLY,
      vertexStorageLevel: StorageLevel =
StorageLevel.MEMORY_ONLY): Graph[VD, Int] =
  {
    val edges = rawEdges.map(p => Edge(p._1, p._2, 1))
    val graph = GraphImpl(edges, defaultValue,
edgeStorageLevel, vertexStorageLevel)
    uniqueEdges match {
      case Some(p) => graph.partitionBy(p).groupEdges((a,
b) => a + b)
      case None => graph
    }
  }
```

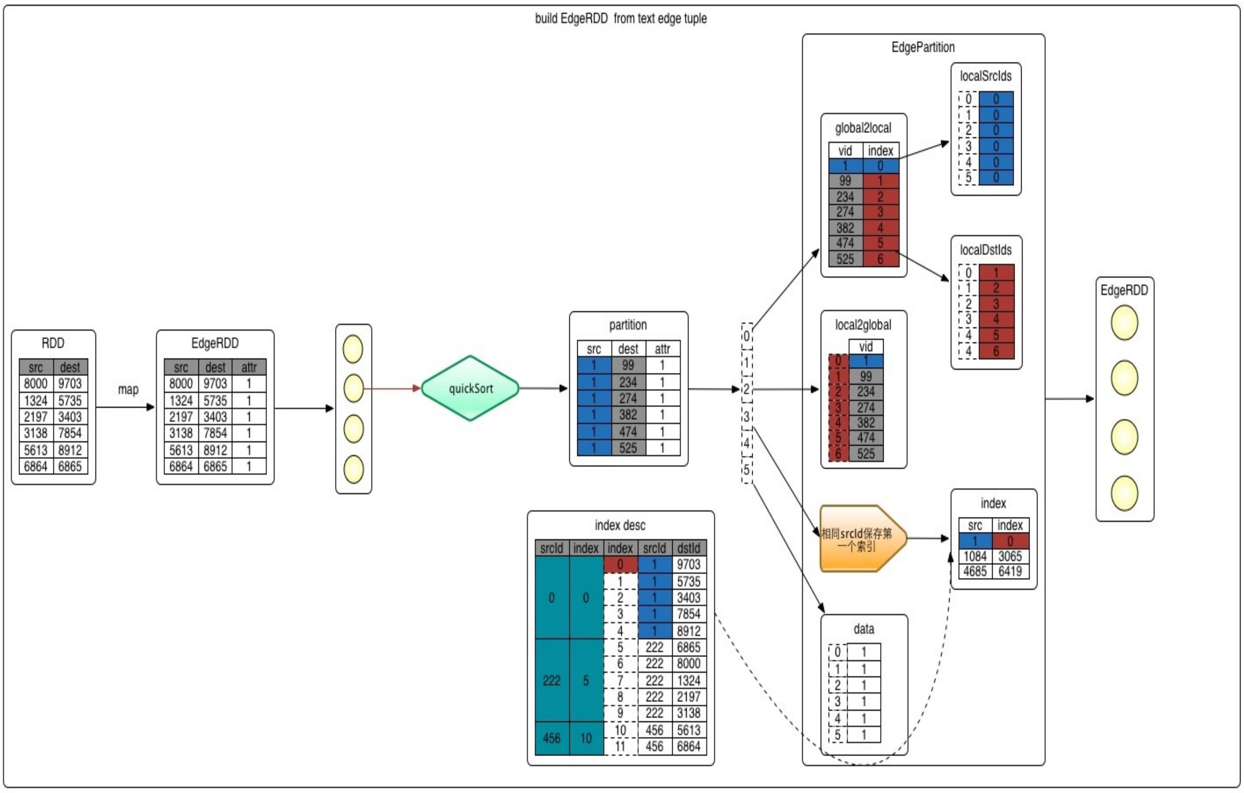
从上面的代码我们知道，不管是根据边构建图还是根据边的两个顶点数据构建，最终都是使用 `GraphImpl` 来构建的，即调用了 `GraphImpl` 的 `apply` 方法。

2 构建图的过程

构建图的过程很简单，分为三步，它们分别是构建边 `EdgeRDD` 、构建顶点 `VertexRDD` 、生成 `Graph` 对象。下面分别介绍这三个步骤。

2.1 构建边 `EdgeRDD`

从源代码看构建边 `EdgeRDD` 也分为三步，下图的例子详细说明了这些步骤。



- **1** 从文件中加载信息，转换成 `tuple` 的形式,即 `(srcId, dstId)`

```
val rawEdgesRdd: RDD[(Long, Long)] =
    sc.textFile(input).filter(s => s !=
"0,0").repartition(partitionNum).map {
      case line =>
        val ss = line.split(",")
        val src = ss(0).toLong
        val dst = ss(1).toLong
        if (src < dst)
          (src, dst)
        else
          (dst, src)
    }.distinct()
```

- **2** 入口，调用 `Graph.fromEdgeTuples(rawEdgesRdd)`

源数据为分割的两个点 `ID` ，把源数据映射成 `Edge(srcId, dstId, attr)` 对象, attr默认为1。这样元数据就构建成了 `RDD[Edge[ED]]` ,如下面的代码

```
val edges = rawEdges.map(p => Edge(p._1, p._2, 1))
```

- **3** 将 `RDD[Edge[ED]]` 进一步转化成 `EdgeRDDImpl[ED, VD]`

第二步构建完 `RDD[Edge[ED]]` 之后，`GraphX` 通过调用 `GraphImpl` 的 `apply` 方法来构建 `Graph` 。

```
val graph = GraphImpl(edges, defaultValue,
edgeStorageLevel, vertexStorageLevel)
def apply[VD: ClassTag, ED: ClassTag](
      edges: RDD[Edge[ED]],
      defaultVertexAttr: VD,
      edgeStorageLevel: StorageLevel,
      vertexStorageLevel: StorageLevel): GraphImpl[VD,
ED] = {
    fromEdgeRDD(EdgeRDD.fromEdges(edges),
defaultVertexAttr, edgeStorageLevel, vertexStorageLevel)
  }
```

在 `apply` 调用 `fromEdgeRDD` 之前，代码会调用 `EdgeRDD.fromEdges(edges)` 将 `RDD[Edge[ED]]` 转化成 `EdgeRDDImpl[ED, VD]` 。

```
def fromEdges[ED: ClassTag, VD: ClassTag](edges:
RDD[Edge[ED]]): EdgeRDDImpl[ED, VD] = {
    val edgePartitions = edges.mapPartitionsWithIndex {
(pid, iter) =>
      val builder = new EdgePartitionBuilder[ED, VD]
      iter.foreach { e =>
        builder.add(e.srcId, e.dstId, e.attr)
      }
      Iterator((pid, builder.toEdgePartition))
    }
    EdgeRDD.fromEdgePartitions(edgePartitions)
  }
```

程序遍历 `RDD[Edge[ED]]` 的每个分区，并调用 `builder.toEdgePartition` 对分区内的边作相应的处理。

```
def toEdgePartition: EdgePartition[ED, VD] = {
    val edgeArray = edges.trim().array
    new Sorter(Edge.edgeArraySortDataFormat[ED])
      .sort(edgeArray, 0, edgeArray.length,
Edge.lexicographicOrdering)
    val localSrcIds = new Array[Int](edgeArray.size)
    val localDstIds = new Array[Int](edgeArray.size)
    val data = new Array[ED](edgeArray.size)
    val index = new
GraphXPrimitiveKeyOpenHashMap[VertexId, Int]
    val global2local = new
GraphXPrimitiveKeyOpenHashMap[VertexId, Int]
    val local2global = new PrimitiveVector[VertexId]
    var vertexAttrs = Array.empty[VD]
    //采用列式存储的方式，节省了空间
    if (edgeArray.length > 0) {
      index.update(edgeArray(0).srcId, 0)
      var currSrcId: VertexId = edgeArray(0).srcId
      var currLocalId = -1
```

```
      var i = 0
      while (i < edgeArray.size) {
        val srcId = edgeArray(i).srcId
        val dstId = edgeArray(i).dstId
        localSrcIds(i) = global2local.changeValue(srcId,
          { currLocalId += 1; local2global += srcId;
currLocalId }, identity)
        localDstIds(i) = global2local.changeValue(dstId,
          { currLocalId += 1; local2global += dstId;
currLocalId }, identity)
        data(i) = edgeArray(i).attr
        //相同顶点srcId中第一个出现的srcId与其下标
        if (srcId != currSrcId) {
          currSrcId = srcId
          index.update(currSrcId, i)
        }
        i += 1
      }
      vertexAttrs = new Array[VD](currLocalId + 1)
    }
    new EdgePartition(
      localSrcIds, localDstIds, data, index,
global2local, local2global.trim().array, vertexAttrs,
      None)
  }
```

- `toEdgePartition` 的第一步就是对边进行排序。

按照 `srcId` 从小到大排序。排序是为了遍历时顺序访问，加快访问速度。采用数组而不是 `Map` ，是因为数组是连续的内存单元，具有原子性，避免了 `Map` 的 `hash` 问题，访问速度快。

- `toEdgePartition` 的第二步就是填充 `localSrcIds,localDstIds, data, index, global2local, local2global, vertexAttrs` 。

数组 `localSrcIds,localDstIds` 中保存的是通过 `global2local.changeValue(srcId/dstId)` 转换而成的分区本地索引。可以通过 `localSrcIds`、`localDstIds` 数组中保存的索引位从 `local2global` 中查到具体的 `VertexId` 。

`global2local` 是一个简单的，`key` 值非负的快速 `hash map` ：`GraphXPrimitiveKeyOpenHashMap` , 保存 `vertextId` 和本地索引的映射关系。`global2local` 中包含当前 `partition` 所有 `srcId` 、 `dstId` 与本地索引的映射关系。

`data` 就是当前分区的 `attr` 属性数组。

我们知道相同的 `srcId` 可能对应不同的 `dstId` 。按照 `srcId` 排序之后，相同的 `srcId` 会出现多行，如上图中的 `index desc` 部分。 `index` 中记录的是相同 `srcId` 中第一个出现的 `srcId` 与其下标。

`local2global` 记录的是所有的 `VertexId` 信息的数组。形如： `srcId,dstId,srcId,dstId,srcId,dstId,srcId,dstId` 。其中会包含相同的 `srcId` 。即：当前分区所有 `vertextId` 的顺序实际值。

我们可以通过根据本地下标取 `VertexId` ，也可以根据 `VertexId` 取本地下标，取相应的属性。

```
// 根据本地下标取VertexId
localSrcIds/localDstIds -> index -> local2global ->
VertexId
// 根据VertexId取本地下标，取属性
VertexId -> global2local -> index -> data -> attr object
```

2.2 构建顶点 `VertexRDD`

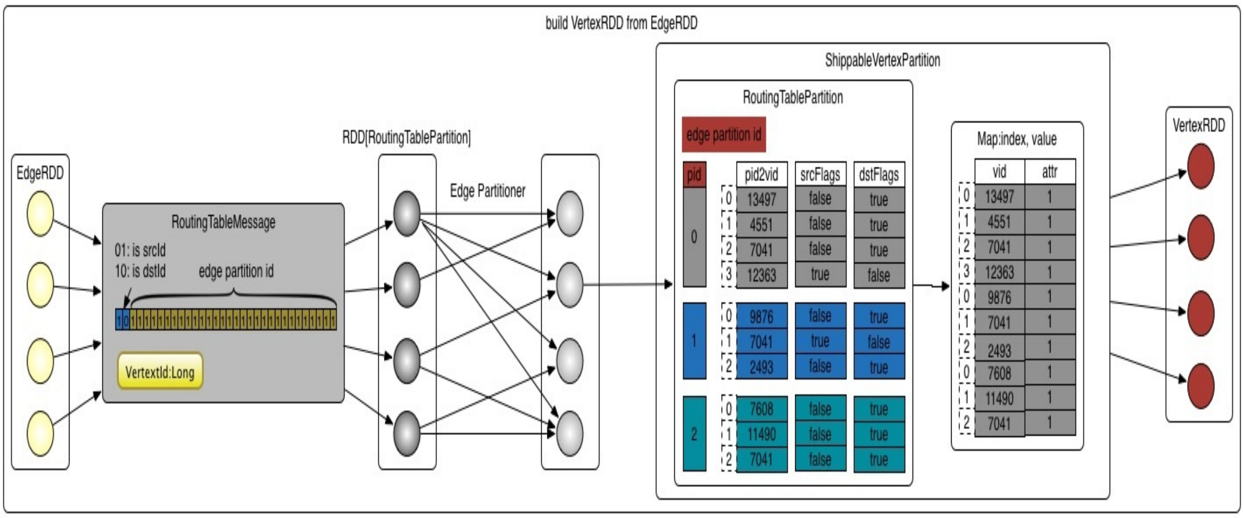
紧接着上面构建边 `RDD` 的代码，我们看看方法 `fromEdgeRDD` 的实现。

```
private def fromEdgeRDD[VD: ClassTag, ED: ClassTag](
      edges: EdgeRDDImpl[ED, VD],
      defaultVertexAttr: VD,
      edgeStorageLevel: StorageLevel,
      vertexStorageLevel: StorageLevel): GraphImpl[VD,
ED] = {
    val edgesCached =
edges.withTargetStorageLevel(edgeStorageLevel).cache()
    val vertices = VertexRDD.fromEdges(edgesCached,
edgesCached.partitions.size, defaultVertexAttr)
      .withTargetStorageLevel(vertexStorageLevel)
    fromExistingRDDs(vertices, edgesCached)
  }
```

从上面的代码我们可以知道，`GraphX` 使用 `VertexRDD.fromEdges` 构建顶点 `VertexRDD` ，当然我们把边 `RDD` 作为参数传入。

```
def fromEdges[VD: ClassTag](
      edges: EdgeRDD[_], numPartitions: Int, defaultVal:
VD): VertexRDD[VD] = {
    //1 创建路由表
    val routingTables = createRoutingTables(edges, new
HashPartitioner(numPartitions))
    //2 根据路由表生成分区对象vertexPartitions
    val vertexPartitions = routingTables.mapPartitions({
routingTableIter =>
      val routingTable =
        if (routingTableIter.hasNext)
routingTableIter.next() else RoutingTablePartition.empty
      Iterator(ShippableVertexPartition(Iterator.empty,
routingTable, defaultVal))
    }, preservesPartitioning = true)
    //3 创建VertexRDDImpl对象
    new VertexRDDImpl(vertexPartitions)
  }
```

构建顶点 `VertexRDD` 的过程分为三步，如上代码中的注释。它的构建过程如下图所示：



- **1** 创建路由表

为了能通过点找到边，每个点需要保存点到边的信息，这些信息保存在 `RoutingTablePartition` 中。

```
private[graphx] def createRoutingTables(
      edges: EdgeRDD[_], vertexPartitioner: Partitioner):
RDD[RoutingTablePartition] = {
    // 将edge partition中的数据转换成RoutingTableMessage类
型，
    val vid2pid =
edges.partitionsRDD.mapPartitions(_.flatMap(

Function.tupled(RoutingTablePartition.edgePartitionToMsgs
)))
  }
```

上述程序首先将边分区中的数据转换成 `RoutingTableMessage` 类型，即 `tuple(VertexId,Int)` 类型。

```
def edgePartitionToMsgs(pid: PartitionID, edgePartition:
EdgePartition[_, _])
    : Iterator[RoutingTableMessage] = {
    val map = new GraphXPrimitiveKeyOpenHashMap[VertexId,
Byte]
    edgePartition.iterator.foreach { e =>
      map.changeValue(e.srcId, 0x1, (b: Byte) => (b |
0x1).toByte)
      map.changeValue(e.dstId, 0x2, (b: Byte) => (b |
0x2).toByte)
    }
    map.iterator.map { vidAndPosition =>
      val vid = vidAndPosition._1
      val position = vidAndPosition._2
      toMessage(vid, pid, position)
    }
  }
//`30-0`比特位表示边分区`ID`,`32-31`比特位表示标志位
private def toMessage(vid: VertexId, pid: PartitionID,
position: Byte): RoutingTableMessage = {
    val positionUpper2 = position << 30
    val pidLower30 = pid & 0x3FFFFFFF
    (vid, positionUpper2 | pidLower30)
  }
```

根据代码，我们可以知道程序使用 `int` 的 `32-31` 比特位表示标志位，即 `01: isSrcId ,10: isDstId` 。 `30-0` 比特位表示边分区 `ID` 。这样做可以节省内存。 `RoutingTableMessage` 表达的信息是：顶点 `id` 和它相关联的边的分区 `id` 是放在一起的,所以任何时候，我们都可以通过 `RoutingTableMessage` 找到顶点关联的边。

- **2** 根据路由表生成分区对象

```
private[graphx] def createRoutingTables(
      edges: EdgeRDD[_], vertexPartitioner: Partitioner):
RDD[RoutingTablePartition] = {
    // 将edge partition中的数据转换成RoutingTableMessage类
型，
    val numEdgePartitions = edges.partitions.size
    vid2pid.partitionBy(vertexPartitioner).mapPartitions(
      iter =>
Iterator(RoutingTablePartition.fromMsgs(numEdgePartitions
, iter)),
      preservesPartitioning = true)
  }
```

我们将第1步生成的 `vid2pid` 按照 `HashPartitioner` 重新分区。我们看看 `RoutingTablePartition.fromMsgs` 方法。

```
 def fromMsgs(numEdgePartitions: Int, iter:
Iterator[RoutingTableMessage])
    : RoutingTablePartition = {
    val pid2vid = Array.fill(numEdgePartitions)(new
PrimitiveVector[VertexId])
    val srcFlags = Array.fill(numEdgePartitions)(new
PrimitiveVector[Boolean])
    val dstFlags = Array.fill(numEdgePartitions)(new
PrimitiveVector[Boolean])
    for (msg <- iter) {
      val vid = vidFromMessage(msg)
      val pid = pidFromMessage(msg)
      val position = positionFromMessage(msg)
      pid2vid(pid) += vid
      srcFlags(pid) += (position & 0x1) != 0
      dstFlags(pid) += (position & 0x2) != 0
    }
    new RoutingTablePartition(pid2vid.zipWithIndex.map {
      case (vids, pid) => (vids.trim().array,
toBitSet(srcFlags(pid)), toBitSet(dstFlags(pid)))
    })
  }
```

该方法从 `RoutingTableMessage` 获取数据，将 `vid`，边 `pid`，`isSrcId/isDstId` 重新封装到 `pid2vid`，`srcFlags`，`dstFlags` 这三个数据结构中。它们表示当前顶点分区中的点在边分区的分布。想象一下，重新分区后，新分区中的点可能来自于不同的边分区，所以一个点要找到边，就需要先确定边的分区号 `pid`，然后在确定的边分区中确定是 `srcId` 还是 `dstId`，这样就找到了边。新分区中保存 `vids.trim().array, toBitSet(srcFlags(pid)), toBitSet(dstFlags(pid))` 这样的记录。这里转换为 `toBitSet` 保存是为了节省空间。

根据上文生成的 `routingTables`，重新封装路由表里的数据结构为 `ShippableVertexPartition`。`ShippableVertexPartition` 会合并相同重复点的属性 `attr` 对象，补全缺失的 `attr` 对象。

```
def apply[VD: ClassTag](
      iter: Iterator[(VertexId, VD)], routingTable:
RoutingTablePartition, defaultVal: VD,
      mergeFunc: (VD, VD) => VD):
ShippableVertexPartition[VD] = {
    val map = new GraphXPrimitiveKeyOpenHashMap[VertexId,
VD]
    // 合并顶点
    iter.foreach { pair =>
      map.setMerge(pair._1, pair._2, mergeFunc)
    }
    // 不全缺失的属性值
    routingTable.iterator.foreach { vid =>
      map.changeValue(vid, defaultVal, identity)
    }
    new ShippableVertexPartition(map.keySet, map._values,
map.keySet.getBitSet, routingTable)
  }
//ShippableVertexPartition定义
ShippableVertexPartition[VD: ClassTag](
val index: VertexIdToIndexMap,
val values: Array[VD],
val mask: BitSet,
val routingTable: RoutingTablePartition)
```

`map` 就是映射 `vertexId->attr` ， `index` 就是顶点集合， `values` 就是顶点集对应的属性集， `mask` 指顶点集的 `BitSet` 。

2.3 生成Graph对象

使用上述构建的 `edgeRDD` 和 `vertexRDD` ，使用 `new GraphImpl(vertices, new ReplicatedVertexView(edges.asInstanceOf[EdgeRDDImpl[ED, VD]]))` 就可以生成 `Graph` 对象。 `ReplicatedVertexView` 是点和边的视图，用来管理运送(`shipping`)顶点属性到 `EdgeRDD` 的分区。当顶点属性改变时，我们需要运送它

们到边分区来更新保存在边分区的顶点属性。 注意，在 `ReplicatedVertexView` 中不要保存一个对边的引用，因为在属性运送等级升级后，这个引用可能会发生改变。

```
class ReplicatedVertexView[VD: ClassTag, ED: ClassTag](
    var edges: EdgeRDDImpl[ED, VD],
    var hasSrcId: Boolean = false,
    var hasDstId: Boolean = false)
```

3 参考文献

【1】Graphx:构建graph和聚合消息

【2】spark源码

GraphX的图运算操作

转换操作

`GraphX` 中的转换操作主要有 `mapVertices`，`mapEdges` 和 `mapTriplets` 三个，它们在 `Graph` 文件中定义，在 `GraphImpl` 文件中实现。下面分别介绍这三个方法。

1 `mapVertices`

`mapVertices` 用来更新顶点属性。从图的构建那章我们知道，顶点属性保存在边分区中，所以我们需要改变的是边分区中的属性。

```
override def mapVertices[VD2: ClassTag]
    (f: (VertexId, VD) => VD2)(implicit eq: VD =:= VD2 =
null): Graph[VD2, ED] = {
    if (eq != null) {
      vertices.cache()
      // 使用方法f处理vertices
      val newVerts =
vertices.mapVertexPartitions(_.map(f)).cache()
      //获得两个不同vertexRDD的不同
      val changedVerts =
vertices.asInstanceOf[VertexRDD[VD2]].diff(newVerts)
      //更新ReplicatedVertexView
      val newReplicatedVertexView =
replicatedVertexView.asInstanceOf[ReplicatedVertexView[VD2
, ED]]
        .updateVertices(changedVerts)
      new GraphImpl(newVerts, newReplicatedVertexView)
    } else {
      GraphImpl(vertices.mapVertexPartitions(_.map(f)),
replicatedVertexView.edges)
    }
  }
```

上面的代码中，当 `VD` 和 `VD2` 类型相同时，我们可以重用没有发生变化的点，否则需要重新创建所有的点。我们分析 `VD` 和 `VD2` 相同的情况，分四步处理。

- 1 使用方法 `f` 处理 `vertices` ,获得新的 `VertexRDD`
- 2 使用在 `VertexRDD` 中定义的 `diff` 方法求出新 `VertexRDD` 和源 `VertexRDD` 的不同

```
override def diff(other: VertexRDD[VD]): VertexRDD[VD] =
{
    val otherPartition = other match {
      case other: VertexRDD[_] if this.partitioner ==
other.partitioner =>
        other.partitionsRDD
      case _ =>

VertexRDD(other.partitionBy(this.partitioner.get)).partit
ionsRDD
    }
    val newPartitionsRDD = partitionsRDD.zipPartitions(
      otherPartition, preservesPartitioning = true
    ) { (thisIter, otherIter) =>
      val thisPart = thisIter.next()
      val otherPart = otherIter.next()
      Iterator(thisPart.diff(otherPart))
    }
    this.withPartitionsRDD(newPartitionsRDD)
  }
```

这个方法首先处理新生成的 `VertexRDD` 的分区，如果它的分区和源 `VertexRDD` 的分区一致，那么直接取出它的 `partitionsRDD` ,否则重新分区后取出它的 `partitionsRDD` 。针对新旧两个 `VertexRDD` 的所有分区，调用 `VertexPartitionBaseOps` 中的 `diff` 方法求得分区的不同。

```
def diff(other: Self[VD]): Self[VD] = {
    //首先判断
    if (self.index != other.index) {
      diff(createUsingIndex(other.iterator))
    } else {
      val newMask = self.mask & other.mask
      var i = newMask.nextSetBit(0)
      while (i >= 0) {
        if (self.values(i) == other.values(i)) {
          newMask.unset(i)
        }
        i = newMask.nextSetBit(i + 1)
      }
      this.withValues(other.values).withMask(newMask)
    }
  }
```

该方法隐藏两个 `VertexRDD` 中相同的顶点信息，得到一个新的 `VertexRDD` 。

- 3 更新 `ReplicatedVertexView`

```
def updateVertices(updates: VertexRDD[VD]):
ReplicatedVertexView[VD, ED] = {
    //生成一个VertexAttributeBlock
    val shippedVerts =
updates.shipVertexAttributes(hasSrcId, hasDstId)
      .setName("ReplicatedVertexView.updateVertices -
shippedVerts %s %s (broadcast)".format(
        hasSrcId, hasDstId))
      .partitionBy(edges.partitioner.get)
    //生成新的边RDD
    val newEdges =
edges.withPartitionsRDD(edges.partitionsRDD.zipPartitions
(shippedVerts) {
      (ePartIter, shippedVertsIter) => ePartIter.map {
        case (pid, edgePartition) =>
          (pid,
edgePartition.updateVertices(shippedVertsIter.flatMap(_._
2.iterator)))
      }
    })
    new ReplicatedVertexView(newEdges, hasSrcId,
hasDstId)
  }
```

`updateVertices` 方法返回一个新的 `ReplicatedVertexView` ,它更新了边分区中包含的顶点属性。我们看看它的实现过程。首先看 `shipVertexAttributes` 方法的调用。调用 `shipVertexAttributes` 方法会生成一个 `VertexAttributeBlock` ，`VertexAttributeBlock` 包含当前分区的顶点属性，这些属性可以在特定的边分区使用。

```
def shipVertexAttributes(
      shipSrc: Boolean, shipDst: Boolean):
Iterator[(PartitionID, VertexAttributeBlock[VD])] = {
    Iterator.tabulate(routingTable.numEdgePartitions) {
pid =>
      val initialSize = if (shipSrc && shipDst)
routingTable.partitionSize(pid) else 64
      val vids = new PrimitiveVector[VertexId]
(initialSize)
      val attrs = new PrimitiveVector[VD](initialSize)
      var i = 0
      routingTable.foreachWithinEdgePartition(pid,
shipSrc, shipDst) { vid =>
        if (isDefined(vid)) {
          vids += vid
          attrs += this(vid)
        }
        i += 1
      }
      //（边分区id，VertexAttributeBlock（顶点id，属性））
      (pid, new VertexAttributeBlock(vids.trim().array,
attrs.trim().array))
    }
  }
```

获得新的顶点属性之后，我们就可以调用 `updateVertices` 更新边中顶点的属性了，如下面代码所示：

```
edgePartition.updateVertices(shippedVertsIter.flatMap(_._
2.iterator))
//更新EdgePartition的属性
def updateVertices(iter: Iterator[(VertexId, VD)]):
EdgePartition[ED, VD] = {
    val newVertexAttrs = new Array[VD]
(vertexAttrs.length)
    System.arraycopy(vertexAttrs, 0, newVertexAttrs, 0,
vertexAttrs.length)
    while (iter.hasNext) {
      val kv = iter.next()
      //global2local获得顶点的本地index
      newVertexAttrs(global2local(kv._1)) = kv._2
    }
    new EdgePartition(
      localSrcIds, localDstIds, data, index,
global2local, local2global, newVertexAttrs,
      activeSet)
  }
```

2 mapEdges

`mapEdges` 用来更新边属性。

```
 override def mapEdges[ED2: ClassTag](
      f: (PartitionID, Iterator[Edge[ED]]) =>
Iterator[ED2]): Graph[VD, ED2] = {
    val newEdges = replicatedVertexView.edges
      .mapEdgePartitions((pid, part) => part.map(f(pid,
part.iterator)))
    new GraphImpl(vertices,
replicatedVertexView.withEdges(newEdges))
  }
```

相比于 `mapVertices` ， `mapEdges` 显然要简单得多，它只需要根据方法 `f` 生成新的 `EdgeRDD` ,然后再初始化即可。

3 `mapTriplets` ：用来更新边属性

`mapTriplets` 用来更新边属性。

```
override def mapTriplets[ED2: ClassTag](
      f: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) =>
Iterator[ED2],
      tripletFields: TripletFields): Graph[VD, ED2] = {
    vertices.cache()
    replicatedVertexView.upgrade(vertices,
tripletFields.useSrc, tripletFields.useDst)
    val newEdges =
replicatedVertexView.edges.mapEdgePartitions { (pid,
part) =>
      part.map(f(pid,
part.tripletIterator(tripletFields.useSrc,
tripletFields.useDst)))
    }
    new GraphImpl(vertices,
replicatedVertexView.withEdges(newEdges))
  }
```

这段代码中， `replicatedVertexView` 调用 `upgrade` 方法修改当前的 `ReplicatedVertexView` ，使调用者可以访问到指定级别的边信息（如仅仅可以读源顶点的属性）。

```
def upgrade(vertices: VertexRDD[VD], includeSrc: Boolean, 
includeDst: Boolean) {
    //判断传递级别
    val shipSrc = includeSrc && !hasSrcId
    val shipDst = includeDst && !hasDstId
    if (shipSrc || shipDst) {
      val shippedVerts: RDD[(Int, 
VertexAttributeBlock[VD])] =
        vertices.shipVertexAttributes(shipSrc, shipDst)
          .setName("ReplicatedVertexView.upgrade(%s, %s)
- shippedVerts %s %s (broadcast)".format(
            includeSrc, includeDst, shipSrc, shipDst))
          .partitionBy(edges.partitioner.get)
      val newEdges = 
edges.withPartitionsRDD(edges.partitionsRDD.zipPartitions
(shippedVerts) {
        (ePartIter, shippedVertsIter) => ePartIter.map {
          case (pid, edgePartition) =>
            (pid, 
edgePartition.updateVertices(shippedVertsIter.flatMap(_._
2.iterator)))
        }
      })
      edges = newEdges
      hasSrcId = includeSrc
      hasDstId = includeDst
    }
  }
```

最后，用 `f` 处理边，生成新的 `RDD` ，最后用新的数据初始化图。

4 总结

调用 `mapVertices` , `mapEdges` 和 `mapTriplets` 时，其内部的结构化索引（ `Structural indices` ）并不会发生变化，它们都重用路由表中的数据。

结构操作

当前的 `GraphX` 仅仅支持一组简单的常用结构性操作。下面是基本的结构性操作列表。

```
class Graph[VD, ED] {
  def reverse: Graph[VD, ED]
  def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
               vpred: (VertexId, VD) => Boolean):
Graph[VD, ED]
  def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD,
ED]
  def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
}
```

下面分别介绍这四种函数的原理。

1 reverse

`reverse` 操作返回一个新的图，这个图的边的方向都是反转的。例如，这个操作可以用来计算反转的PageRank。因为反转操作没有修改顶点或者边的属性或者改变边的数量，所以我们可以 在不移动或者复制数据的情况下有效地实现它。

```
override def reverse: Graph[VD, ED] = {
    new GraphImpl(vertices.reverseRoutingTables(),
replicatedVertexView.reverse())
}
def reverse(): ReplicatedVertexView[VD, ED] = {
    val newEdges = edges.mapEdgePartitions((pid, part) =>
part.reverse)
    new ReplicatedVertexView(newEdges, hasDstId,
hasSrcId)
}
//EdgePartition中的reverse
def reverse: EdgePartition[ED, VD] = {
    val builder = new ExistingEdgePartitionBuilder[ED,
VD](
      global2local, local2global, vertexAttrs, activeSet,
size)
    var i = 0
    while (i < size) {
      val localSrcId = localSrcIds(i)
      val localDstId = localDstIds(i)
      val srcId = local2global(localSrcId)
      val dstId = local2global(localDstId)
      val attr = data(i)
      //将源顶点和目标顶点换位置
      builder.add(dstId, srcId, localDstId, localSrcId,
attr)
      i += 1
    }
    builder.toEdgePartition
  }
```

2 subgraph

`subgraph` 操作利用顶点和边的判断式（ `predicates` ），返回的图仅仅包含满足顶点判断式的顶点、满足边判断式的边以及满足顶点判断式的 `triple` 。 `subgraph` 操作可以用于很多场景，如获取 感兴趣的顶点和边组成的图或者获取清除断开连接后的图。

```
override def subgraph(
      epred: EdgeTriplet[VD, ED] => Boolean = x => true,
      vpred: (VertexId, VD) => Boolean = (a, b) => true):
Graph[VD, ED] = {
    vertices.cache()
    // 过滤vertices, 重用partitioner和索引
    val newVerts =
vertices.mapVertexPartitions(_.filter(vpred))
    // 过滤 triplets
    replicatedVertexView.upgrade(vertices, true, true)
    val newEdges =
replicatedVertexView.edges.filter(epred, vpred)
    new GraphImpl(newVerts,
replicatedVertexView.withEdges(newEdges))
  }
```

该代码显示， `subgraph` 方法的实现分两步：先过滤 `VertexRDD` ，然后再过滤 `EdgeRDD` 。如上，过滤 `VertexRDD` 比较简单，我们重点看过滤 `EdgeRDD` 的过程。

```
def filter(
      epred: EdgeTriplet[VD, ED] => Boolean,
      vpred: (VertexId, VD) => Boolean): EdgeRDDImpl[ED,
VD] = {
    mapEdgePartitions((pid, part) => part.filter(epred,
vpred))
  }
//EdgePartition中的filter方法
def filter(
      epred: EdgeTriplet[VD, ED] => Boolean,
```

```
      vpred: (VertexId, VD) => Boolean):
EdgePartition[ED, VD] = {
    val builder = new ExistingEdgePartitionBuilder[ED,
VD](
      global2local, local2global, vertexAttrs, activeSet)
    var i = 0
    while (i < size) {
      // The user sees the EdgeTriplet, so we can't reuse
it and must create one per edge.
      val localSrcId = localSrcIds(i)
      val localDstId = localDstIds(i)
      val et = new EdgeTriplet[VD, ED]
      et.srcId = local2global(localSrcId)
      et.dstId = local2global(localDstId)
      et.srcAttr = vertexAttrs(localSrcId)
      et.dstAttr = vertexAttrs(localDstId)
      et.attr = data(i)
      if (vpred(et.srcId, et.srcAttr) && vpred(et.dstId,
et.dstAttr) && epred(et)) {
        builder.add(et.srcId, et.dstId, localSrcId,
localDstId, et.attr)
      }
      i += 1
    }
    builder.toEdgePartition
  }
```

因为用户可以看到 `EdgeTriplet` 的信息，所以我们不能重用 `EdgeTriplet` ，需要重新创建一个，然后在用 `epred` 函数处理。这里 `localSrcIds,localDstIds,local2global` 等前文均有介绍，在此不再赘述。

3 mask

`mask` 操作构造一个子图，这个子图包含输入图中包含的顶点和边。它的实现很简单，顶点和边均做 `inner join` 操作即可。这个操作可以和 `subgraph` 操作相结合，基于另外一个相关图的特征去约束一个图。

```
override def mask[VD2: ClassTag, ED2: ClassTag] (
      other: Graph[VD2, ED2]): Graph[VD, ED] = {
    val newVerts = vertices.innerJoin(other.vertices) {
(vid, v, w) => v }
    val newEdges =
replicatedVertexView.edges.innerJoin(other.edges) { (src,
dst, v, w) => v }
    new GraphImpl(newVerts,
replicatedVertexView.withEdges(newEdges))
  }
```

4 groupEdges

`groupEdges` 操作合并多重图中的并行边(如顶点对之间重复的边)。在大量的应用程序中，并行的边可以合并（它们的权重合并）为一条边从而降低图的大小。

```
 override def groupEdges(merge: (ED, ED) => ED):
Graph[VD, ED] = {
    val newEdges =
replicatedVertexView.edges.mapEdgePartitions(
      (pid, part) => part.groupEdges(merge))
    new GraphImpl(vertices,
replicatedVertexView.withEdges(newEdges))
  }
 def groupEdges(merge: (ED, ED) => ED): EdgePartition[ED,
VD] = {
     val builder = new ExistingEdgePartitionBuilder[ED,
VD](
       global2local, local2global, vertexAttrs,
activeSet)
     var currSrcId: VertexId =
```

```
null.asInstanceOf[VertexId]
      var currDstId: VertexId =
null.asInstanceOf[VertexId]
      var currLocalSrcId = -1
      var currLocalDstId = -1
      var currAttr: ED = null.asInstanceOf[ED]
      // 迭代处理所有的边
      var i = 0
      while (i < size) {
        //如果源顶点和目的顶点都相同
        if (i > 0 && currSrcId == srcIds(i) && currDstId
== dstIds(i)) {
          // 合并属性
          currAttr = merge(currAttr, data(i))
        } else {
          // This edge starts a new run of edges
          if (i > 0) {
            // 添加到builder中
            builder.add(currSrcId, currDstId,
currLocalSrcId, currLocalDstId, currAttr)
          }
          // Then start accumulating for a new run
          currSrcId = srcIds(i)
          currDstId = dstIds(i)
          currLocalSrcId = localSrcIds(i)
          currLocalDstId = localDstIds(i)
          currAttr = data(i)
        }
        i += 1
      }
      if (size > 0) {
        builder.add(currSrcId, currDstId, currLocalSrcId,
currLocalDstId, currAttr)
      }
      builder.toEdgePartition
    }
```

在图构建那章我们说明过，存储的边按照源顶点 `id` 排过序，所以上面的代码可以通过一次迭代完成对所有相同边的处理。

5 应用举例

```
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L,
("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L,
("istoica", "prof")),
                       (4L, ("peter", "student"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),
Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"),
Edge(5L, 7L, "pi"),
                       Edge(4L, 0L, "student"),
Edge(5L, 0L, "colleague")))
// Define a default user in case there are relationship
with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
// Notice that there is a user 0 (for which we have no
information) connected to users
// 4 (peter) and 5 (franklin).
graph.triplets.map(
    triplet => triplet.srcAttr._1 + " is the " +
triplet.attr + " of " + triplet.dstAttr._1
  ).collect.foreach(println(_))
// Remove missing vertices as well as the edges to
connected to them
```

```
val validGraph = graph.subgraph(vpred = (id, attr) =>
attr._2 != "Missing")
// The valid subgraph will disconnect users 4 and 5 by
removing user 0
validGraph.vertices.collect.foreach(println(_))
validGraph.triplets.map(
    triplet => triplet.srcAttr._1 + " is the " +
triplet.attr + " of " + triplet.dstAttr._1
  ).collect.foreach(println(_))

/ Run Connected Components
val ccGraph = graph.connectedComponents() // No longer
contains missing field
// Remove missing vertices as well as the edges to
connected to them
val validGraph = graph.subgraph(vpred = (id, attr) =>
attr._2 != "Missing")
// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

6 参考文献

【1】spark源码

关联操作

在许多情况下，有必要将外部数据加入到图中。例如，我们可能有额外的用户属性需要合并到已有的图中或者我们可能想从一个图中取出顶点特征加入到另外一个图中。这些任务可以用 `join` 操作完成。 主要的 `join` 操作如下所示。

```
class Graph[VD, ED] {
  def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
    : Graph[VD, ED]
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
}
```

`joinVertices` 操作 `join` 输入 `RDD` 和顶点，返回一个新的带有顶点特征的图。这些特征是通过在连接顶点的结果上使用用户定义的 `map` 函数获得的。没有匹配的顶点保留其原始值。 下面详细地来分析这两个函数。

1 joinVertices

```
def joinVertices[U: ClassTag](table: RDD[(VertexId, U)])(mapFunc: (VertexId, VD, U) => VD)
    : Graph[VD, ED] = {
    val uf = (id: VertexId, data: VD, o: Option[U]) => {
      o match {
        case Some(u) => mapFunc(id, data, u)
        case None => data
      }
    }
    graph.outerJoinVertices(table)(uf)
  }
```

我们可以看到，`joinVertices` 的实现是通过 `outerJoinVertices` 来实现的。这是因为 `join` 本来就是 `outer join` 的一种特例。

2 outerJoinVertices

```
override def outerJoinVertices[U: ClassTag, VD2:
ClassTag]
      (other: RDD[(VertexId, U)])
      (updateF: (VertexId, VD, Option[U]) => VD2)
      (implicit eq: VD =:= VD2 = null): Graph[VD2, ED] =
{
    if (eq != null) {
      vertices.cache()
      // updateF preserves type, so we can use
incremental replication
      val newVerts = vertices.leftJoin(other)
(updateF).cache()
      val changedVerts =
vertices.asInstanceOf[VertexRDD[VD2]].diff(newVerts)
      val newReplicatedVertexView =
replicatedVertexView.asInstanceOf[ReplicatedVertexView[VD2
, ED]]
        .updateVertices(changedVerts)
      new GraphImpl(newVerts, newReplicatedVertexView)
    } else {
      // updateF does not preserve type, so we must re-
replicate all vertices
      val newVerts = vertices.leftJoin(other)(updateF)
      GraphImpl(newVerts, replicatedVertexView.edges)
    }
  }
```

通过以上的代码我们可以看到，如果 `updateF` 不改变类型，我们只需要创建改变的顶点即可，否则我们要重新创建所有的顶点。我们讨论不改变类型的情况。这种情况分三步。

- 1 修改顶点属性值

```
val newVerts = vertices.leftJoin(other)(updateF).cache()
```

这一步会用顶点 `RDD` `join` 传入的 `RDD` ，然后用 `updateF` 作用 `joinRDD` 中的所有顶点，改变它们的值。

- 2 找到发生改变的顶点

```
  val changedVerts =
vertices.asInstanceOf[VertexRDD[VD2]].diff(newVerts)
```

- 3 更新newReplicatedVertexView中边分区中的顶点属性

```
val newReplicatedVertexView =
replicatedVertexView.asInstanceOf[ReplicatedVertexView[VD2
, ED]]
        .updateVertices(changedVerts)
```

第2、3两步的源码已经在转换操作中详细介绍。

聚合操作

`GraphX` 中提供的聚合操作有 `aggregateMessages` 、 `collectNeighborIds` 和 `collectNeighbors` 三个，其中 `aggregateMessages` 在 `GraphImpl` 中实现， `collectNeighborIds` 和 `collectNeighbors` 在 `GraphOps` 中实现。下面分别介绍这几个方法。

1 `aggregateMessages`

1.1 `aggregateMessages` 接口

`aggregateMessages` 是 `GraphX` 最重要的 `API` ，用于替换 `mapReduceTriplets` 。目前 `mapReduceTriplets` 最终也是通过 `aggregateMessages` 来实现的。它主要功能是向邻边发消息，合并邻边收到的消息，返回 `messageRDD` 。 `aggregateMessages` 的接口如下：

```
def aggregateMessages[A: ClassTag](
      sendMsg: EdgeContext[VD, ED, A] => Unit,
      mergeMsg: (A, A) => A,
      tripletFields: TripletFields = TripletFields.All)
    : VertexRDD[A] = {
    aggregateMessagesWithActiveSet(sendMsg, mergeMsg,
tripletFields, None)
  }
```

该接口有三个参数，分别为发消息函数，合并消息函数以及发消息的方向。

- `sendMsg` ： 发消息函数

```
private def sendMsg(ctx: EdgeContext[KCoreVertex, Int,
Map[Int, Int]]): Unit = {
    ctx.sendToDst(Map(ctx.srcAttr.preKCore -> -1,
ctx.srcAttr.curKCore -> 1))
    ctx.sendToSrc(Map(ctx.dstAttr.preKCore -> -1,
ctx.dstAttr.curKCore -> 1))
}
```

- `mergeMsg` ：合并消息函数

该函数用于在 `Map` 阶段每个 `edge` 分区中每个点收到的消息合并，并且它还用于 `reduce` 阶段，合并不同分区的消息。合并 `vertexId` 相同的消息。

- `tripletFields` ：定义发消息的方向

1.2 `aggregateMessages` 处理流程

`aggregateMessages` 方法分为 `Map` 和 `Reduce` 两个阶段，下面我们分别就这两个阶段说明。

1.2.1 Map阶段

从入口函数进入 `aggregateMessagesWithActiveSet` 函数，该函数首先使用 `VertexRDD[VD]` 更新 `replicatedVertexView` , 只更新其中 `vertexRDD` 中 `attr` 对象。如构建图中介绍的， `replicatedVertexView` 是点和边的视图，点的属性有变化，要更新边中包含的点的 `attr` 。

```
replicatedVertexView.upgrade(vertices,
tripletFields.useSrc, tripletFields.useDst)
val view = activeSetOpt match {
    case Some((activeSet, _)) =>
      //返回只包含活跃顶点的replicatedVertexView
      replicatedVertexView.withActiveSet(activeSet)
    case None =>
      replicatedVertexView
}
```

程序然后会对`replicatedVertexView`的`edgeRDD`做`mapPartitions`操作，所有的操作都在每个边分区的迭代中完成，如下面的代码：

```
 val preAgg =
view.edges.partitionsRDD.mapPartitions(_.flatMap {
      case (pid, edgePartition) =>
        // 选择 scan 方法
        val activeFraction =
edgePartition.numActives.getOrElse(0) /
edgePartition.indexSize.toFloat
        activeDirectionOpt match {
          case Some(EdgeDirection.Both) =>
            if (activeFraction < 0.8) {

edgePartition.aggregateMessagesIndexScan(sendMsg,
mergeMsg, tripletFields,
                EdgeActiveness.Both)
            } else {

edgePartition.aggregateMessagesEdgeScan(sendMsg,
mergeMsg, tripletFields,
                EdgeActiveness.Both)
            }
          case Some(EdgeDirection.Either) =>
```

```
edgePartition.aggregateMessagesEdgeScan(sendMsg,
mergeMsg, tripletFields,
              EdgeActiveness.Either)
          case Some(EdgeDirection.Out) =>
            if (activeFraction < 0.8) {

edgePartition.aggregateMessagesIndexScan(sendMsg,
mergeMsg, tripletFields,
                EdgeActiveness.SrcOnly)
            } else {

edgePartition.aggregateMessagesEdgeScan(sendMsg,
mergeMsg, tripletFields,
                EdgeActiveness.SrcOnly)
            }
          case Some(EdgeDirection.In) =>

edgePartition.aggregateMessagesEdgeScan(sendMsg,
mergeMsg, tripletFields,
              EdgeActiveness.DstOnly)
          case _ => // None

edgePartition.aggregateMessagesEdgeScan(sendMsg,
mergeMsg, tripletFields,
              EdgeActiveness.Neither)
        }
    })
```

在分区内，根据 `activeFraction` 的大小选择是进入 `aggregateMessagesEdgeScan` 还是 `aggregateMessagesIndexScan` 处理。`aggregateMessagesEdgeScan` 会顺序地扫描所有的边，而 `aggregateMessagesIndexScan` 会先过滤源顶点索引，然后在扫描。我们重点去分析 `aggregateMessagesEdgeScan` 。

```
def aggregateMessagesEdgeScan[A: ClassTag](
      sendMsg: EdgeContext[VD, ED, A] => Unit,
      mergeMsg: (A, A) => A,
      tripletFields: TripletFields,
      activeness: EdgeActiveness): Iterator[(VertexId,
A)] = {
    var ctx = new AggregatingEdgeContext[VD, ED, A]
(mergeMsg, aggregates, bitset)
    var i = 0
    while (i < size) {
      val localSrcId = localSrcIds(i)
      val srcId = local2global(localSrcId)
      val localDstId = localDstIds(i)
      val dstId = local2global(localDstId)
      val srcAttr = if (tripletFields.useSrc)
vertexAttrs(localSrcId) else null.asInstanceOf[VD]
      val dstAttr = if (tripletFields.useDst)
vertexAttrs(localDstId) else null.asInstanceOf[VD]
      ctx.set(srcId, dstId, localSrcId, localDstId,
srcAttr, dstAttr, data(i))
      sendMsg(ctx)
      i += 1
    }
```

该方法由两步组成，分别是获得顶点相关信息，以及发送消息。

- 获取顶点相关信息

在前文介绍 `edge partition` 时，我们知道它包含 `localSrcIds,localDstIds`, `data`, `index`, `global2local`, `local2global`, `vertexAttrs` 这几个重要的数据结构。其中 `localSrcIds,localDstIds` 分别表示源顶点、目的顶点在当前分区中的索引。所以我们可以遍历 `localSrcIds`,根据其下标去 `localSrcIds` 中拿到 `srcId` 在全局 `local2global` 中的索引，最后拿到 `srcId` 。通过 `vertexAttrs` 拿到顶点属性。通过 `data` 拿到边属性。

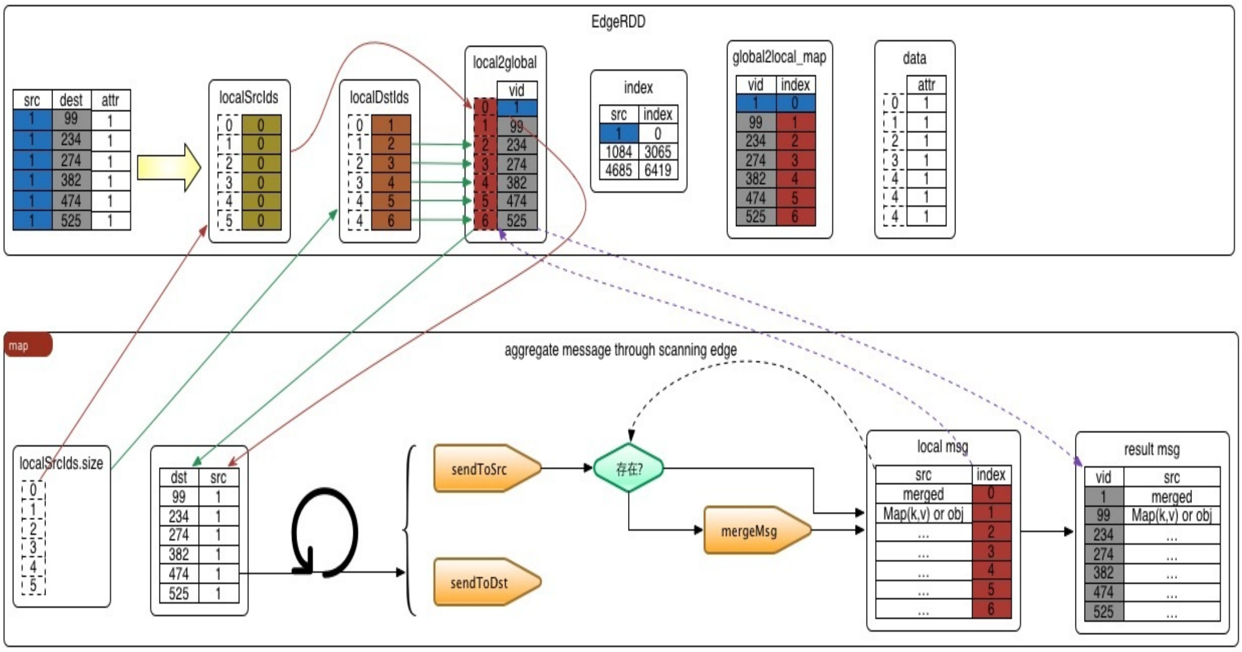
- 发送消息

发消息前会根据接口中定义的 `tripletFields` ，拿到发消息的方向。发消息的过程就是遍历到一条边，向 `localSrcIds/localDstIds` 中添加数据，如果 `localSrcIds/localDstIds` 中已经存在该数据，则执行合并函数 `mergeMsg` 。

```
 override def sendToSrc(msg: A) {
    send(_localSrcId, msg)
  }
  override def sendToDst(msg: A) {
    send(_localDstId, msg)
  }
  @inline private def send(localId: Int, msg: A) {
    if (bitset.get(localId)) {
      aggregates(localId) = mergeMsg(aggregates(localId),
msg)
    } else {
      aggregates(localId) = msg
      bitset.set(localId)
    }
  }
```

每个点之间在发消息的时候是独立的，即：点单纯根据方向，向以相邻点的以 `localId` 为下标的数组中插数据，互相独立，可以并行运行。 `Map` 阶段最后返回消息 `RDD` `messages: RDD[(VertexId, VD2)]`

`Map` 阶段的执行流程如下例所示：



1.2.2 Reduce阶段

`Reduce` 阶段的实现就是调用下面的代码

```
vertices.aggregateUsingIndex(preAgg, mergeMsg)
override def aggregateUsingIndex[VD2: ClassTag](
      messages: RDD[(VertexId, VD2)], reduceFunc: (VD2, 
VD2) => VD2): VertexRDD[VD2] = {
    val shuffled = 
messages.partitionBy(this.partitioner.get)
    val parts = partitionsRDD.zipPartitions(shuffled, 
true) { (thisIter, msgIter) =>
      thisIter.map(_.aggregateUsingIndex(msgIter, 
reduceFunc))
    }
    this.withPartitionsRDD[VD2](parts)
  }
```

上面的代码通过两步实现。

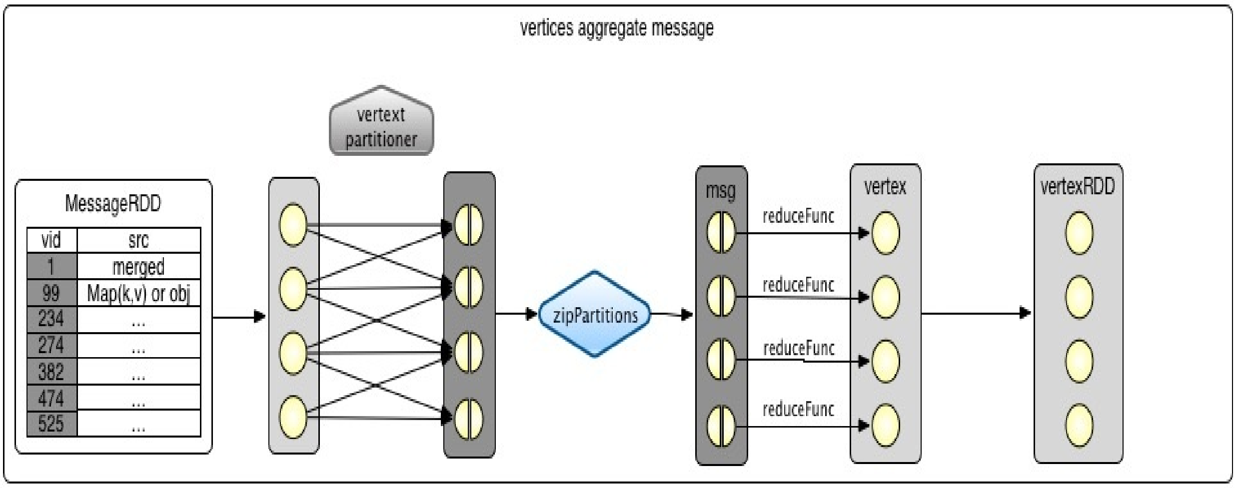
- 1 对 `messages` 重新分区，分区器使用 `VertexRDD` 的 `partitioner` 。然后使用 `zipPartitions` 合并两个分区。
- 2 对等合并 `attr` , 聚合函数使用传入的 `mergeMsg` 函数

```
def aggregateUsingIndex[VD2: ClassTag](
      iter: Iterator[Product2[VertexId, VD2]],
      reduceFunc: (VD2, VD2) => VD2): Self[VD2] = {
    val newMask = new BitSet(self.capacity)
    val newValues = new Array[VD2](self.capacity)
    iter.foreach { product =>
      val vid = product._1
      val vdata = product._2
      val pos = self.index.getPos(vid)
      if (pos >= 0) {
        if (newMask.get(pos)) {
          newValues(pos) = reduceFunc(newValues(pos),
vdata)
        } else { // otherwise just store the new value
          newMask.set(pos)
          newValues(pos) = vdata
        }
      }
    }
    this.withValues(newValues).withMask(newMask)
  }
```

根据传参，我们知道上面的代码迭代的是 `messagePartition` ，并不是每个节点都会收到消息，所以 `messagePartition` 集合最小，迭代速度会快。

这段代码表示，我们根据 `vetexId` 从 `index` 中取到其下标 `pos` ,再根据下标，从 `values` 中取到 `attr` ，存在 `attr` 就用 `mergeMsg` 合并 `attr` ，不存在就直接赋值。

`Reduce` 阶段的过程如下图所示：



1.3 举例

下面的例子计算比用户年龄大的追随者（即 `followers` ）的平均年龄。

```
// Import random graph generation library
import org.apache.spark.graphx.util.GraphGenerators
// Create a graph with "age" as the vertex property.
Here we use a random graph for simplicity.
val graph: Graph[Double, Int] =
  GraphGenerators.logNormalGraph(sc, numVertices =
100).mapVertices( (id, _) => id.toDouble )
// Compute the number of older followers and their total
age
val olderFollowers: VertexRDD[(Int, Double)] =
graph.aggregateMessages[(Int, Double)](
  triplet => { // Map Function
    if (triplet.srcAttr > triplet.dstAttr) {
      // Send message to destination vertex containing
counter and age
      triplet.sendToDst(1, triplet.srcAttr)
    }
  },
  // Add counter and age
  (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce Function
)
// Divide total age by number of older followers to get
average age of older followers
val avgAgeOfOlderFollowers: VertexRDD[Double] =
  olderFollowers.mapValues( (id, value) => value match {
case (count, totalAge) => totalAge / count } )
// Display the results
avgAgeOfOlderFollowers.collect.foreach(println(_))
```

2 `collectNeighbors`

该方法的作用是收集每个顶点的邻居顶点的顶点 `id` 和顶点属性。

```
 def collectNeighbors(edgeDirection: EdgeDirection):
VertexRDD[Array[(VertexId, VD)]] = {
    val nbrs = edgeDirection match {
      case EdgeDirection.Either =>
        graph.aggregateMessages[Array[(VertexId, VD)]](
          ctx => {
            ctx.sendToSrc(Array((ctx.dstId,
ctx.dstAttr)))
            ctx.sendToDst(Array((ctx.srcId,
ctx.srcAttr)))
          },
          (a, b) => a ++ b, TripletFields.All)
      case EdgeDirection.In =>
        graph.aggregateMessages[Array[(VertexId, VD)]](
          ctx => ctx.sendToDst(Array((ctx.srcId,
ctx.srcAttr))),
          (a, b) => a ++ b, TripletFields.Src)
      case EdgeDirection.Out =>
        graph.aggregateMessages[Array[(VertexId, VD)]](
          ctx => ctx.sendToSrc(Array((ctx.dstId,
ctx.dstAttr))),
          (a, b) => a ++ b, TripletFields.Dst)
      case EdgeDirection.Both =>
        throw new SparkException("collectEdges does not
support EdgeDirection.Both. Use" +
          "EdgeDirection.Either instead.")
    }
    graph.vertices.leftJoin(nbrs) { (vid, vdata, nbrsOpt)
=>
      nbrsOpt.getOrElse(Array.empty[(VertexId, VD)])
    }
  }
```

从上面的代码中，第一步是根据`EdgeDirection`来确定调用哪个`aggregateMessages`实现聚合操作。我们用满足条件`EdgeDirection.Either`的情况来说明。可以看到`aggregateMessages`的方式消息的函数为：

```
ctx => {
          ctx.sendToSrc(Array((ctx.dstId, ctx.dstAttr)))
          ctx.sendToDst(Array((ctx.srcId, ctx.srcAttr)))
       },
```

这个函数在处理每条边时都会同时向源顶点和目的顶点发送消息，消息内容分别为`（目的顶点id，目的顶点属性）`、`（源顶点id，源顶点属性）`。为什么会这样处理呢？我们知道，每条边都由两个顶点组成，对于这个边，我需要向源顶点发送目的顶点的信息来记录它们之间的邻居关系，同理向目的顶点发送源顶点的信息来记录它们之间的邻居关系。

`Merge`函数是一个集合合并操作，它合并同同一个顶点对应的所有目的顶点的信息。如下所示：

```
(a, b) => a ++ b
```

通过`aggregateMessages`获得包含邻居关系信息的`VertexRDD`后，把它和现有的`vertices`作`join`操作，得到每个顶点的邻居消息。

3 `collectNeighborIds`

该方法的作用是收集每个顶点的邻居顶点的顶点`id`。它的实现和`collectNeighbors`非常相同。

```
def collectNeighborIds(edgeDirection: EdgeDirection):
VertexRDD[Array[VertexId]] = {
    val nbrs =
      if (edgeDirection == EdgeDirection.Either) {
        graph.aggregateMessages[Array[VertexId]](
          ctx => { ctx.sendToSrc(Array(ctx.dstId));
ctx.sendToDst(Array(ctx.srcId)) },
          _ ++ _, TripletFields.None)
      } else if (edgeDirection == EdgeDirection.Out) {
        graph.aggregateMessages[Array[VertexId]](
          ctx => ctx.sendToSrc(Array(ctx.dstId)),
          _ ++ _, TripletFields.None)
      } else if (edgeDirection == EdgeDirection.In) {
        graph.aggregateMessages[Array[VertexId]](
          ctx => ctx.sendToDst(Array(ctx.srcId)),
          _ ++ _, TripletFields.None)
      } else {
        throw new SparkException("It doesn't make sense
to collect neighbor ids without a " +
          "direction. (EdgeDirection.Both is not
supported; use EdgeDirection.Either instead.)")
      }
    graph.vertices.leftZipJoin(nbrs) { (vid, vdata,
nbrsOpt) =>
      nbrsOpt.getOrElse(Array.empty[VertexId])
    }
  }
```

和 `collectNeighbors` 的实现不同的是， `aggregateMessages` 函数中的 `sendMsg` 函数只发送顶点 `Id` 到源顶点和目的顶点。其它的实现基本一致。

```
ctx => { ctx.sendToSrc(Array(ctx.dstId));
ctx.sendToDst(Array(ctx.srcId)) }
```

4 参考文献

【1】Graphx:构建graph和聚合消息

【2】spark源码

缓存操作

在 `Spark` 中， `RDD` 默认是不缓存的。为了避免重复计算，当需要多次利用它们时，我们必须显示地缓存它们。 `GraphX` 中的图也有相同的方式。当利用到图多次时，确保首先访问 `Graph.cache()` 方法。

在迭代计算中，为了获得最佳的性能，不缓存可能是必须的。默认情况下，缓存的 `RDD` 和图会一直保留在内存中直到因为内存压力迫使它们以 `LRU` 的顺序删除。对于迭代计算，先前的迭代的中间结果将填充到缓存 中。虽然它们最终会被删除，但是保存在内存中的不需要的数据将会减慢垃圾回收。只有中间结果不需要，不缓存它们是更高效的。然而，因为图是由多个 `RDD` 组成的，正确的不持久化它们是困难的。对于迭代计算，我们建议使用 `Pregel API` ，它可以正确的不持久化中间结果。

`GraphX` 中的缓存操作有 `cache` , `persist` , `unpersist` 和 `unpersistVertices` 。它们的接口分别是：

```
def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
def cache(): Graph[VD, ED]
def unpersist(blocking: Boolean = true): Graph[VD, ED]
def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
```

Pregel API

图本身是递归数据结构，顶点的属性依赖于它们邻居的属性，这些邻居的属性又依赖于自己邻居的属性。所以许多重要的图算法都是迭代的重新计算每个顶点的属性，直到满足某个确定的条件。 一系列的图并发(`graph-parallel`)抽象已经被提出来用来表达这些迭代算法。 `GraphX`公开了一个类似`Pregel`的操作，它是广泛使用的`Pregel`和`GraphLab`抽象的一个融合。

`GraphX`中实现的这个更高级的`Pregel`操作是一个约束到图拓扑的批量同步（`bulk-synchronous`）并行消息抽象。 `Pregel`操作者执行一系列的超步（`super steps`），在这些步骤中，顶点从 之前的超步中接收进入(`inbound`)消息的总和，为顶点属性计算一个新的值，然后在以后的超步中发送消息到邻居顶点。不像`Pregel`而更像`GraphLab`，消息通过边`triplet`的一个函数被并行计算， 消息的计算既会访问源顶点特征也会访问目的顶点特征。在超步中，没有收到消息的顶点会被跳过。当没有消息遗留时， `Pregel`操作停止迭代并返回最终的图。

注意，与标准的`Pregel`实现不同的是， `GraphX`中的顶点仅仅能发送信息给邻居顶点，并且可以利用用户自定义的消息函数并行地构造消息。这些限制允许对`GraphX`进行额外的优化。

下面的代码是`pregel`的具体实现。

```
def apply[VD: ClassTag, ED: ClassTag, A: ClassTag]
     (graph: Graph[VD, ED],
      initialMsg: A,
      maxIterations: Int = Int.MaxValue,
      activeDirection: EdgeDirection =
EdgeDirection.Either)
     (vprog: (VertexId, VD, A) => VD,
      sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexId,
A)],
      mergeMsg: (A, A) => A)
    : Graph[VD, ED] =
  {
    var g = graph.mapVertices((vid, vdata) => vprog(vid,
vdata, initialMsg)).cache()
    // 计算消息
    var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
    var activeMessages = messages.count()
    // 迭代
    var prevG: Graph[VD, ED] = null
    var i = 0
    while (activeMessages > 0 && i < maxIterations) {
      // 接收消息并更新顶点
      prevG = g
      g = g.joinVertices(messages)(vprog).cache()
      val oldMessages = messages
      // 发送新消息
      messages = g.mapReduceTriplets(
        sendMsg, mergeMsg, Some((oldMessages,
activeDirection))).cache()
      activeMessages = messages.count()
      i += 1
    }
    g
  }
```

1 pregel计算模型

`Pregel` 计算模型中有三个重要的函数，分别是 `vertexProgram` 、 `sendMessage` 和 `messageCombiner` 。

- `vertexProgram` ：用户定义的顶点运行程序。它作用于每一个顶点，负责接收进来的信息，并计算新的顶点值。
- `sendMsg` ：发送消息
- `mergeMsg` ：合并消息

我们具体分析它的实现。根据代码可以知道，这个实现是一个迭代的过程。在开始迭代之前，先完成一些初始化操作：

```
var g = graph.mapVertices((vid, vdata) => vprog(vid, vdata, initialMsg)).cache()
// 计算消息
var messages = g.mapReduceTriplets(sendMsg, mergeMsg)
var activeMessages = messages.count()
```

程序首先用 `vprog` 函数处理图中所有的顶点，生成新的图。然后用生成的图调用聚合操作（ `mapReduceTriplets` ，实际的实现是我们前面章节讲到的 `aggregateMessagesWithActiveSet` 函数）获取聚合后的消息。 `activeMessages` 指 `messages` 这个 `VertexRDD` 中的顶点数。

下面就开始迭代操作了。在迭代内部，分为二步。

- 1 接收消息，并更新顶点

```
g = g.joinVertices(messages)(vprog).cache()
  //joinVertices的定义
  def joinVertices[U: ClassTag](table: RDD[(VertexId, U)])
(mapFunc: (VertexId, VD, U) => VD)
      : Graph[VD, ED] = {
      val uf = (id: VertexId, data: VD, o: Option[U]) => {
        o match {
          case Some(u) => mapFunc(id, data, u)
          case None => data
        }
      }
      graph.outerJoinVertices(table)(uf)
    }
```

这一步实际上是使用 `outerJoinVertices` 来更新顶点属性。 `outerJoinVertices` 在关联操作中有详细介绍。

- 2 发送新消息

```
messages = g.mapReduceTriplets(
         sendMsg, mergeMsg, Some((oldMessages,
activeDirection))).cache()
```

注意，在上面的代码中， `mapReduceTriplets` 多了一个参数 `Some((oldMessages, activeDirection))` 。这个参数的作用是：它使我们在发送新的消息时，会忽略掉那些两端都没有接收到消息的边，减少计算量。

2 pregel实现最短路径

```
import org.apache.spark.graphx._
import org.apache.spark.graphx.util.GraphGenerators
val graph: Graph[Long, Double] =
  GraphGenerators.logNormalGraph(sc, numVertices =
100).mapEdges(e => e.attr.toDouble)
val sourceId: VertexId = 42 // The ultimate source
// 初始化图
val initialGraph = graph.mapVertices((id, _) => if (id ==
sourceId) 0.0 else Double.PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), //
Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr)
{
      Iterator((triplet.dstId, triplet.srcAttr +
triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a,b) => math.min(a,b) // Merge Message
  )
println(sssp.vertices.collect.mkString("\n"))
```

上面的例子中，`Vertex Program` 函数定义如下：

```
(id, dist, newDist) => math.min(dist, newDist)
```

这个函数的定义显而易见，当两个消息来的时候，取它们当中路径的最小值。同理 `Merge Message` 函数也是同样的含义。

`Send Message` 函数中，会首先比较 `triplet.srcAttr + triplet.attr` 和 `triplet.dstAttr`，即比较加上边的属性后，这个值是否小于目的节点的属性，如果小于，则发送消息到目的顶点。

3 参考文献

【1】spark源码

广度优先遍历

```
val graph = GraphLoader.edgeListFile(sc,
"graphx/data/test_graph.txt")

val root: VertexId = 1
val initialGraph = graph.mapVertices((id, _) => if (id ==
root) 0.0 else
Double.PositiveInfinity)

val vprog = { (id: VertexId, attr: Double, msg: Double)
=> math.min(attr,msg) }

val sendMessage = { (triplet: EdgeTriplet[Double, Int])
=>
        var iter:Iterator[(VertexId, Double)] =
Iterator.empty
        val isSrcMarked = triplet.srcAttr !=
Double.PositiveInfinity
        val isDstMarked = triplet.dstAttr !=
Double.PositiveInfinity
        if(!(isSrcMarked && isDstMarked)){
               if(isSrcMarked){
                iter =
Iterator((triplet.dstId,triplet.srcAttr+1))
              }else{
                iter =
Iterator((triplet.srcId,triplet.dstAttr+1))
               }
        }
        iter
}

val reduceMessage = { (a: Double, b: Double) =>
```

```
math.min(a,b) }

val bfs = initialGraph.pregel(Double.PositiveInfinity,
20)(vprog, sendMessage, reduceMessage)

println(bfs.vertices.collect.mkString("\n"))
```

单源最短路径

```
import scala.reflect.ClassTag

import org.apache.spark.graphx._

/**
 * Computes shortest paths to the given set of landmark
vertices, returning a graph where each
 * vertex attribute is a map containing the shortest-path
distance to each reachable landmark.
 */
object ShortestPaths {
  /** Stores a map from the vertex id of a landmark to
the distance to that landmark. */
  type SPMap = Map[VertexId, Int]

  private def makeMap(x: (VertexId, Int)*) = Map(x: _*)

  private def incrementMap(spmap: SPMap): SPMap =
spmap.map { case (v, d) => v -> (d + 1) }

  private def addMaps(spmap1: SPMap, spmap2: SPMap):
SPMap =
    (spmap1.keySet ++ spmap2.keySet).map {
      k => k -> math.min(spmap1.getOrElse(k,
Int.MaxValue), spmap2.getOrElse(k, Int.MaxValue))
    }.toMap

  /**
   * Computes shortest paths to the given set of landmark
vertices.
   *
   * @tparam ED the edge attribute type (not used in the
```

```
computation)
   *
   * @param graph the graph for which to compute the
shortest paths
   * @param landmarks the list of landmark vertex ids.
Shortest paths will be computed to each
   * landmark.
   *
   * @return a graph where each vertex attribute is a map
containing the shortest-path distance to
   * each reachable landmark vertex.
   */
  def run[VD, ED: ClassTag](graph: Graph[VD, ED],
landmarks: Seq[VertexId]): Graph[SPMap, ED] = {
    val spGraph = graph.mapVertices { (vid, attr) =>
      if (landmarks.contains(vid)) makeMap(vid -> 0) else
makeMap()
    }

    val initialMessage = makeMap()

    def vertexProgram(id: VertexId, attr: SPMap, msg:
SPMap): SPMap = {
      addMaps(attr, msg)
    }

    def sendMessage(edge: EdgeTriplet[SPMap, _]):
Iterator[(VertexId, SPMap)] = {
      val newAttr = incrementMap(edge.dstAttr)
      if (edge.srcAttr != addMaps(newAttr, edge.srcAttr))
Iterator((edge.srcId, newAttr))
      else Iterator.empty
    }

    Pregel(spGraph, initialMessage)(vertexProgram,
sendMessage, addMaps)
```

```
    }
}
```

连通图

```
import scala.reflect.ClassTag

import org.apache.spark.graphx._

/** Connected components algorithm. */
object ConnectedComponents {
  /**
   * Compute the connected component membership of each
vertex and return a graph with the vertex
   * value containing the lowest vertex id in the
connected component containing that vertex.
   *
   * @tparam VD the vertex attribute type (discarded in
the computation)
   * @tparam ED the edge attribute type (preserved in the
computation)
   * @param graph the graph for which to compute the
connected components
   * @param maxIterations the maximum number of
iterations to run for
   * @return a graph with vertex attributes containing
the smallest vertex in each
   *         connected component
   */
  def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD,
ED],
                                      maxIterations:
Int): Graph[VertexId, ED] = {
    require(maxIterations > 0, s"Maximum of iterations
must be greater than 0," +
      s" but got ${maxIterations}")
```

```
    val ccGraph = graph.mapVertices { case (vid, _) =>
vid }
    def sendMessage(edge: EdgeTriplet[VertexId, ED]):
Iterator[(VertexId, VertexId)] = {
      if (edge.srcAttr < edge.dstAttr) {
        Iterator((edge.dstId, edge.srcAttr))
      } else if (edge.srcAttr > edge.dstAttr) {
        Iterator((edge.srcId, edge.dstAttr))
      } else {
        Iterator.empty
      }
    }
    val initialMessage = Long.MaxValue
    val pregelGraph = Pregel(ccGraph, initialMessage,
      maxIterations, EdgeDirection.Either)(
      vprog = (id, attr, msg) => math.min(attr, msg),
      sendMsg = sendMessage,
      mergeMsg = (a, b) => math.min(a, b))
    ccGraph.unpersist()
    pregelGraph
  } // end of connectedComponents

  /**
   * Compute the connected component membership of each
vertex and return a graph with the vertex
   * value containing the lowest vertex id in the
connected component containing that vertex.
   *
   * @tparam VD the vertex attribute type (discarded in
the computation)
   * @tparam ED the edge attribute type (preserved in the
computation)
   * @param graph the graph for which to compute the
connected components
   * @return a graph with vertex attributes containing
```

```
the smallest vertex in each
   *          connected component
   */
  def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD,
ED]): Graph[VertexId, ED] = {
    run(graph, Int.MaxValue)
  }
}
```

三角计数

```
import scala.reflect.ClassTag

import org.apache.spark.graphx._

/**
 * Compute the number of triangles passing through each vertex.
 *
 * The algorithm is relatively straightforward and can be computed in three steps:
 *
 * <ul>
 * <li> Compute the set of neighbors for each vertex</li>
 * <li> For each edge compute the intersection of the sets and send the count to both vertices.</li>
 * <li> Compute the sum at each vertex and divide by two since each triangle is counted twice.</li>
 * </ul>
 *
 * There are two implementations.  The default `TriangleCount.run` implementation first removes
 * self cycles and canonicalizes the graph to ensure that the following conditions hold:
 * <ul>
 * <li> There are no self edges</li>
 * <li> All edges are oriented src > dst</li>
 * <li> There are no duplicate edges</li>
 * </ul>
 * However, the canonicalization procedure is costly as it requires repartitioning the graph.
 * If the input data is already in "canonical form" with self cycles removed then the
```

```
 * `TriangleCount.runPreCanonicalized` should be used
instead.
 *
 * {{{
 * val canonicalGraph = graph.mapEdges(e =>
1).removeSelfEdges().canonicalizeEdges()
 * val counts =
TriangleCount.runPreCanonicalized(canonicalGraph).vertice
s
 * }}}
 *
 */
object TriangleCount {

  def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD,
ED]): Graph[Int, ED] = {
    // Transform the edge data something cheap to shuffle
and then canonicalize
    val canonicalGraph = graph.mapEdges(e =>
true).removeSelfEdges().convertToCanonicalEdges()
    // Get the triangle counts
    val counters =
runPreCanonicalized(canonicalGraph).vertices
    // Join them bath with the original graph
    graph.outerJoinVertices(counters) { (vid, _,
optCounter: Option[Int]) =>
      optCounter.getOrElse(0)
    }
  }


  def runPreCanonicalized[VD: ClassTag, ED: ClassTag]
(graph: Graph[VD, ED]): Graph[Int, ED] = {
    // Construct set representations of the neighborhoods
    val nbrSets: VertexRDD[VertexSet] =
```

```
graph.collectNeighborIds(EdgeDirection.Either).mapValues
{ (vid, nbrs) =>
        val set = new VertexSet(nbrs.length)
        var i = 0
        while (i < nbrs.length) {
          // prevent self cycle
          if (nbrs(i) != vid) {
            set.add(nbrs(i))
          }
          i += 1
        }
        set
      }

    // join the sets with the graph
    val setGraph: Graph[VertexSet, ED] =
graph.outerJoinVertices(nbrSets) {
      (vid, _, optSet) => optSet.getOrElse(null)
    }

    // Edge function computes intersection of smaller
vertex with larger vertex
    def edgeFunc(ctx: EdgeContext[VertexSet, ED, Int]) {
      val (smallSet, largeSet) = if (ctx.srcAttr.size <
ctx.dstAttr.size) {
        (ctx.srcAttr, ctx.dstAttr)
      } else {
        (ctx.dstAttr, ctx.srcAttr)
      }
      val iter = smallSet.iterator
      var counter: Int = 0
      while (iter.hasNext) {
        val vid = iter.next()
        if (vid != ctx.srcId && vid != ctx.dstId &&
largeSet.contains(vid)) {
          counter += 1
```

```
          }
        }
        ctx.sendToSrc(counter)
        ctx.sendToDst(counter)
      }

      // compute the intersection along edges
      val counters: VertexRDD[Int] =
setGraph.aggregateMessages(edgeFunc, _ + _)
      // Merge counters with the graph and divide by two
since each triangle is counted twice
      graph.outerJoinVertices(counters) { (_, _,
optCounter: Option[Int]) =>
        val dblCount = optCounter.getOrElse(0)
        // This algorithm double counts each triangle so
the final count should be even
        require(dblCount % 2 == 0, "Triangle count resulted
in an invalid number of triangles.")
        dblCount / 2
      }
    }
}
```

PageRank

```
import scala.language.postfixOps
import scala.reflect.ClassTag

import org.apache.spark.graphx._
import org.apache.spark.internal.Logging

/**
 * PageRank algorithm implementation. There are two
implementations of PageRank implemented.
 *
 * The first implementation uses the standalone [[Graph]]
interface and runs PageRank
 * for a fixed number of iterations:
 * {{{
 * var PR = Array.fill(n)( 1.0 )
 * val oldPR = Array.fill(n)( 1.0 )
 * for( iter <- 0 until numIter ) {
 *   swap(oldPR, PR)
 *   for( i <- 0 until n ) {
 *     PR[i] = alpha + (1 - alpha) * inNbrs[i].map(j =>
oldPR[j] / outDeg[j]).sum
 *   }
 * }
 * }}}
 *
 * The second implementation uses the [[Pregel]]
interface and runs PageRank until
 * convergence:
 *
 * {{{
 * var PR = Array.fill(n)( 1.0 )
 * val oldPR = Array.fill(n)( 0.0 )
```

```
 * while( max(abs(PR - oldPr)) > tol ) {
 *   swap(oldPR, PR)
 *   for( i <- 0 until n if abs(PR[i] - oldPR[i]) > tol )
{
 *     PR[i] = alpha + (1 - \alpha) * inNbrs[i].map(j =>
oldPR[j] / outDeg[j]).sum
 *   }
 * }
 * }}}
 *
 * `alpha` is the random reset probability (typically
0.15), `inNbrs[i]` is the set of
 * neighbors which link to `i` and `outDeg[j]` is the out
degree of vertex `j`.
 *
 * Note that this is not the "normalized" PageRank and as
a consequence pages that have no
 * inlinks will have a PageRank of alpha.
 */
object PageRank extends Logging {


  /**
   * Run PageRank for a fixed number of iterations
returning a graph
   * with vertex attributes containing the PageRank and
edge
   * attributes the normalized edge weight.
   *
   * @tparam VD the original vertex attribute (not used)
   * @tparam ED the original edge attribute (not used)
   *
   * @param graph the graph on which to compute PageRank
   * @param numIter the number of iterations of PageRank
to run
   * @param resetProb the random reset probability
```

```
(alpha)
   *
   * @return the graph containing with each vertex
containing the PageRank and each edge
   *         containing the normalized weight.
   */
  def run[VD: ClassTag, ED: ClassTag](graph: Graph[VD,
ED], numIter: Int,
    resetProb: Double = 0.15): Graph[Double, Double] =
  {
    runWithOptions(graph, numIter, resetProb)
  }

  /**
   * Run PageRank for a fixed number of iterations
returning a graph
   * with vertex attributes containing the PageRank and
edge
   * attributes the normalized edge weight.
   *
   * @tparam VD the original vertex attribute (not used)
   * @tparam ED the original edge attribute (not used)
   *
   * @param graph the graph on which to compute PageRank
   * @param numIter the number of iterations of PageRank
to run
   * @param resetProb the random reset probability
(alpha)
   * @param srcId the source vertex for a Personalized
Page Rank (optional)
   *
   * @return the graph containing with each vertex
containing the PageRank and each edge
   *         containing the normalized weight.
   *
   */
```

```
  def runWithOptions[VD: ClassTag, ED: ClassTag](
      graph: Graph[VD, ED], numIter: Int, resetProb:
Double = 0.15,
      srcId: Option[VertexId] = None): Graph[Double,
Double] =
  {
    require(numIter > 0, s"Number of iterations must be
greater than 0," +
      s" but got ${numIter}")
    require(resetProb >= 0 && resetProb <= 1, s"Random
reset probability must belong" +
      s" to [0, 1], but got ${resetProb}")

    val personalized = srcId isDefined
    val src: VertexId = srcId.getOrElse(-1L)

    // Initialize the PageRank graph with each edge
attribute having
    // weight 1/outDegree and each vertex with attribute
resetProb.
    // When running personalized pagerank, only the
source vertex
    // has an attribute resetProb. All others are set to
0.
    var rankGraph: Graph[Double, Double] = graph
      // Associate the degree with each vertex
      .outerJoinVertices(graph.outDegrees) { (vid, vdata,
deg) => deg.getOrElse(0) }
      // Set the weight on the edges based on the degree
      .mapTriplets( e => 1.0 / e.srcAttr,
TripletFields.Src )
      // Set the vertex attributes to the initial
pagerank values
      .mapVertices { (id, attr) =>
        if (!(id != src && personalized)) resetProb else
0.0
```

```
      }

    def delta(u: VertexId, v: VertexId): Double = { if (u
== v) 1.0 else 0.0 }

    var iteration = 0
    var prevRankGraph: Graph[Double, Double] = null
    while (iteration < numIter) {
      rankGraph.cache()

      // Compute the outgoing rank contributions of each
vertex, perform local preaggregation, and
      // do the final aggregation at the receiving
vertices. Requires a shuffle for aggregation.
      val rankUpdates =
rankGraph.aggregateMessages[Double](
        ctx => ctx.sendToDst(ctx.srcAttr * ctx.attr), _ +
_, TripletFields.Src)

      // Apply the final rank updates to get the new
ranks, using join to preserve ranks of vertices
      // that didn't receive a message. Requires a
shuffle for broadcasting updated ranks to the
      // edge partitions.
      prevRankGraph = rankGraph
      val rPrb = if (personalized) {
        (src: VertexId, id: VertexId) => resetProb *
delta(src, id)
      } else {
        (src: VertexId, id: VertexId) => resetProb
      }

      rankGraph = rankGraph.joinVertices(rankUpdates) {
        (id, oldRank, msgSum) => rPrb(src, id) + (1.0 -
resetProb) * msgSum
      }.cache()
```

```
      rankGraph.edges.foreachPartition(x => {}) // also
materializes rankGraph.vertices
      logInfo(s"PageRank finished iteration $iteration.")
      prevRankGraph.vertices.unpersist(false)
      prevRankGraph.edges.unpersist(false)

      iteration += 1
    }

    rankGraph
  }

  /**
   * Run a dynamic version of PageRank returning a graph
with vertex attributes containing the
   * PageRank and edge attributes containing the
normalized edge weight.
   *
   * @tparam VD the original vertex attribute (not used)
   * @tparam ED the original edge attribute (not used)
   *
   * @param graph the graph on which to compute PageRank
   * @param tol the tolerance allowed at convergence
(smaller => more accurate).
   * @param resetProb the random reset probability
(alpha)
   *
   * @return the graph containing with each vertex
containing the PageRank and each edge
   *         containing the normalized weight.
   */
  def runUntilConvergence[VD: ClassTag, ED: ClassTag](
    graph: Graph[VD, ED], tol: Double, resetProb: Double
= 0.15): Graph[Double, Double] =
  {
```

```
      runUntilConvergenceWithOptions(graph, tol,
resetProb)
  }

  /**
   * Run a dynamic version of PageRank returning a graph
with vertex attributes containing the
   * PageRank and edge attributes containing the
normalized edge weight.
   *
   * @tparam VD the original vertex attribute (not used)
   * @tparam ED the original edge attribute (not used)
   *
   * @param graph the graph on which to compute PageRank
   * @param tol the tolerance allowed at convergence
(smaller => more accurate).
   * @param resetProb the random reset probability
(alpha)
   * @param srcId the source vertex for a Personalized
Page Rank (optional)
   *
   * @return the graph containing with each vertex
containing the PageRank and each edge
   *         containing the normalized weight.
   */
  def runUntilConvergenceWithOptions[VD: ClassTag, ED:
ClassTag](
      graph: Graph[VD, ED], tol: Double, resetProb:
Double = 0.15,
      srcId: Option[VertexId] = None): Graph[Double,
Double] =
  {
    require(tol >= 0, s"Tolerance must be no less than 0,
but got ${tol}")
    require(resetProb >= 0 && resetProb <= 1, s"Random
reset probability must belong" +
```

```
      s" to [0, 1], but got ${resetProb}")

    val personalized = srcId.isDefined
    val src: VertexId = srcId.getOrElse(-1L)

    // Initialize the pagerankGraph with each edge
attribute
    // having weight 1/outDegree and each vertex with
attribute 1.0.
    val pagerankGraph: Graph[(Double, Double), Double] =
graph
      // Associate the degree with each vertex
      .outerJoinVertices(graph.outDegrees) {
        (vid, vdata, deg) => deg.getOrElse(0)
      }
      // Set the weight on the edges based on the degree
      .mapTriplets( e => 1.0 / e.srcAttr )
      // Set the vertex attributes to (initialPR, delta =
0)
      .mapVertices { (id, attr) =>
        if (id == src) (resetProb,
Double.NegativeInfinity) else (0.0, 0.0)
      }
      .cache()

    // Define the three functions needed to implement
PageRank in the GraphX
    // version of Pregel
    def vertexProgram(id: VertexId, attr: (Double,
Double), msgSum: Double): (Double, Double) = {
      val (oldPR, lastDelta) = attr
      val newPR = oldPR + (1.0 - resetProb) * msgSum
      (newPR, newPR - oldPR)
    }

    def personalizedVertexProgram(id: VertexId, attr:
```

```
(Double, Double),
      msgSum: Double): (Double, Double) = {
      val (oldPR, lastDelta) = attr
      var teleport = oldPR
      val delta = if (src==id) 1.0 else 0.0
      teleport = oldPR*delta

      val newPR = teleport + (1.0 - resetProb) * msgSum
      val newDelta = if (lastDelta ==
Double.NegativeInfinity) newPR else newPR - oldPR
      (newPR, newDelta)
    }

    def sendMessage(edge: EdgeTriplet[(Double, Double),
Double]) = {
      if (edge.srcAttr._2 > tol) {
        Iterator((edge.dstId, edge.srcAttr._2 *
edge.attr))
      } else {
        Iterator.empty
      }
    }

    def messageCombiner(a: Double, b: Double): Double = a
+ b

    // The initial message received by all vertices in
PageRank
    val initialMessage = if (personalized) 0.0 else
resetProb / (1.0 - resetProb)

    // Execute a dynamic version of Pregel.
    val vp = if (personalized) {
      (id: VertexId, attr: (Double, Double), msgSum:
Double) =>
        personalizedVertexProgram(id, attr, msgSum)
```

```
    } else {
      (id: VertexId, attr: (Double, Double), msgSum:
Double) =>
        vertexProgram(id, attr, msgSum)
    }

    Pregel(pagerankGraph, initialMessage, activeDirection
= EdgeDirection.Out)(
      vp, sendMessage, messageCombiner)
      .mapVertices((vid, attr) => attr._1)
  } // end of deltaPageRank

}
```